SAND2017-5214R

FPGA Authentication Methods

US-UK Collaboration for Treaty Verification

5 April 2017

Jay Brotz, US Sandia National Laboratories

Ross Hymel, US Sandia National Laboratories

Ratish Punnoose, US Sandia National Laboratories

Tom Mannos, US Sandia National Laboratories

Neil Grant, UK Atomic Weapons Establishment

Neil Evans, UK Atomic Weapons Establishment

Revision 0

Contents

1. Introduction

One of the greatest challenges facing designers of equipment to be used in a nuclear arms control treaty is how to convince the other party in the treaty to trust its results and functionality. Whether the host provides equipment meant to prove treaty obligations and the inspector needs to gain that trust (commonly referred to as authentication), or the inspector provides this equipment and the host needs to gain this trust (commonly considered to be included in certification), one party generally has higher confidence in the equipment at the start of a treaty regime and the other party needs to gain that confidence prior to use. While we focus on authentication in this document—that is, the inspector gaining confidence in host-provided equipment—our conclusions will likely apply to host certification of inspector-provided equipment.

The greatest challenge of equipment authentication generally lies in the element of the equipment with the highest degree of complexity. In many cases, this is the equipment's logic processor. From a development standpoint, the easiest implementation option for a logic processor is usually a general purpose processor, which could be a microprocessor with an operating system or a microcontroller without an operating system, for example, running custom software. The tools for developing functional processing elements using these platforms are quite mature and usable, yielding correct, complex functionality in a short amount of time. However, these tools are also made to be as flexible as possible. Flexibility in a design environment implies extraneous functionality and unused interfaces, which are generally unwanted in a design to be authenticated. On the other end of the flexibility spectrum is discrete digital logic, using integrated circuits that house logic gates, flip flops, timers, counters, and other simple functions. While this limits extraneous functionality and unused interfaces, it quickly becomes unfeasible for designs of any complexity, and also becomes harder to inspect as the chip count increases. An application specific integrated circuit (ASIC) could turn a large number of discrete logic chips into a single chip with the same ability to limit extraneous functionality and interfaces, but at a cost that is prohibitively high for research and development concerns, and also, likely, for treaty use.

We believe that programmable logic chips, specifically field programmable gate arrays (FPGAs), are a good balance between the easy but untrustable software on a general purpose processor and the trustable but infeasible (for discrete logic chips) or prohibitively expensive (for ASICs) custom static logic designs, as illustrated in Figure 1. An FPGA is a single packaged integrated circuit that can be programmed to arrange its internal structure to manifest any digital circuit that is required. It does this by having a large number of primitive logic elements with a programmable routing layer. By changing the routes, or wiring, between these atomic elements, any digital circuit can be realized, as long as the FPGA contains enough logic elements. The hardware, development tools, and standard languages for describing the circuits to be programmed are nearly as inexpensive and easy to use as typical software applications. The ability to limit extraneous functionality and interfaces in FPGA applications is far greater than in general purpose processors running custom software.

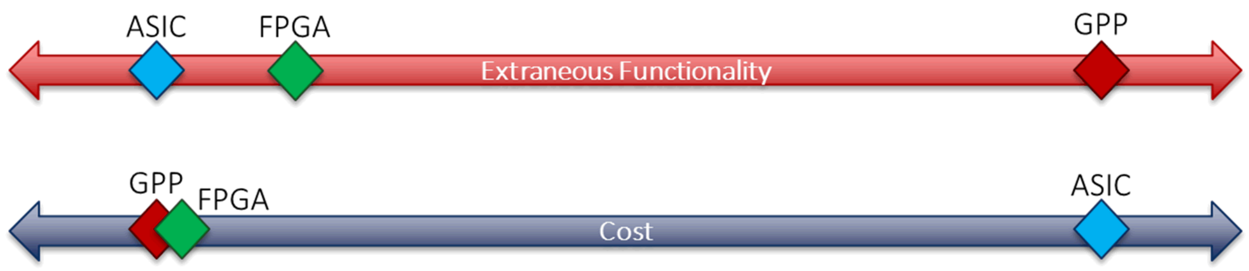


Figure 1. Qualitative Comparison of FPGA to ASIC and General Purpose Processor (GPP)

While FPGAs represent an opportunity for trustable complex logic implementations, the processes by which a treaty party can gain that trust have not been studied to date. In this work, we have outlined a framework for considering this problem, which includes: 1) the elaboration of a generic FPGA development cycle from requirements through operational system, 2) the identification of opportunities for authentication at various points in the development cycle, 3) the prioritization of those authentication methods to create a research plan, 4) the development of a relevant basis system using an FPGA to support evaluations of these methods, and 5) the evaluation of the methods on the basis system. In this report, we discuss the first four parts of this framework, along with the evaluation criteria that will be used in the evaluations. A future report will contain the results of these evaluations with conclusions and recommendations.

1.1 Document Overview

This document contains the identification and prioritization of potential authentication methods for FPGAs and FPGA systems. These methods were evaluated on a basis system containing an FPGA as the primary processing element. This work was conducted in the FPGA Authentication Collaboration between Sandia National Laboratories in the U.S. and the Atomic Weapons Establishment in the UK.

Chapter 2 discusses the FPGA development cycle, the identification of opportunities for authentication, and the prioritization of those methods. Chapter 3 discusses the basis system on which our evaluations will be based. Chapter 4 discusses the evaluation criteria that will be used in those evaluations. Appendices contain supporting materials, such as an elaboration of the authentication methods identified and the design drawings of the basis system.

This document has not been standardized to either American spelling or British spelling of words. Sections written by the US members of the collaboration use American spelling and sections written by the UK members of the collaboration use British spelling.

2. FPGA Authentication Methods and Prioritization

In order to understand the full scope of potential authentication methods, the full development cycle of a generic FPGA system used for treaty verification is represented in Figure 2 through Figure 5. Figure 2 represents the high-level flow from requirements, which we assume for this study are agreed through a negotiation process and are part of the treaty, through design and operation. The application design flow is represented in Figure 3 and the hardware design flow is represented in Figure 4. Finally, the operation flow is represented in Figure 5. In each diagram, a box represents a process and a tilted parallelogram represents an artifact (which could be a design document or a piece of hardware, for example) that can be an input to one process and an output from another. The points within this development flow that are accessible to the inspector have been highlighted in red. In Figure 3 and

Figure 4, only artifacts are used for authentication, whereas in Figure 5 authentication methods also happen during a process (Load Bitstream[1]).

As shown in Figure 2, development is based on requirements that, for this project, are assumed to be codified in a treaty or agreement between the parties. For an FPGA system, the application development (that is, the development of the logic or behavior of the system) is conducted in parallel with the hardware development. The output of each development phase is combined in the system operation phase by loading the bitstream onto the FPGA within a printed circuit board (PCB) containing support electronics and the necessary interfaces. The system is then operated.

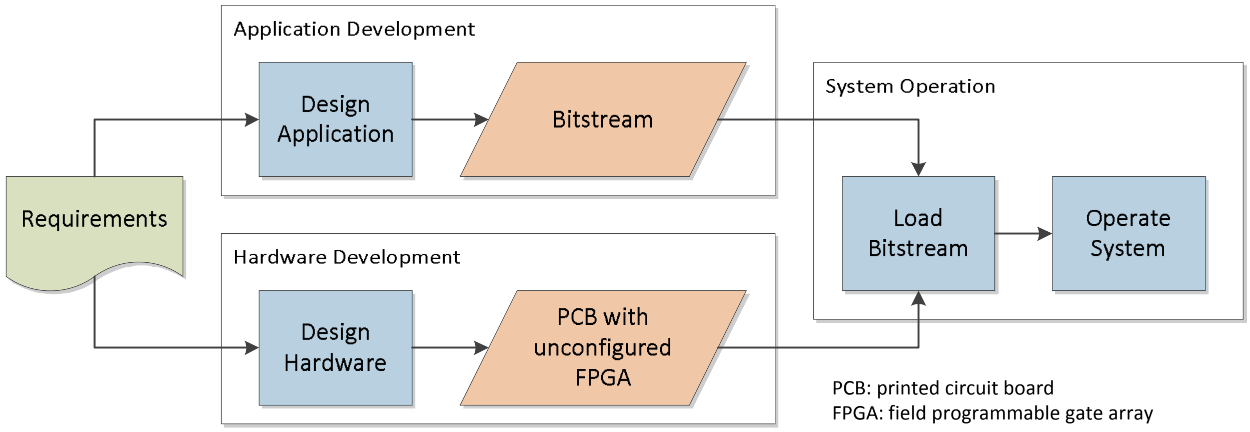


Figure 2. FPGA System Development Flow

As shown in Figure 3, application development for an FPGA is abstracted into a design flow that is supported by an FPGA toolchain. It begins with the creation of register transfer level (RTL) code, usually in Verilog or VHDL, that fully describes the electrical circuit that will be implemented in the FPGA. Then a tool, often provided by the FPGA manufacturer, is used to synthesize the RTL into a netlist, which is a standard representation of any electrical circuit. Successive tool steps then add more detail and specificity to the netlist by mapping (identifying primitive electronic components to be used), placing and optimizing (identifying the specific elements on the FPGA's logic fabric to be used and optimizing the locations of those elements for timing), and routing (identifying the exact path of the wires connecting each of the primitive elements into the circuit specified). This final netlist contains all of the specificity needed to configure the FPGA. This final netlist is then encoded into a bitstream that is used directly by the FPGA to configure its routing when loaded. This is a serial flow and each artifact created by a process can be used in an authentication method, either to compare that artifact to the requirements, or to compare it to an earlier artifact.

[1] The Load Bitstream process is different dependent on the type of FPGA: for Flash-based FPGAs, it means programming (or "burning") the FPGA into on-chip non-volatile memory; for static random-access memory (SRAM)-based FPGAs, it means programming the external configuration memory or inserting programmed external memory into a socket on the board, so that the bitstream is loaded onto the FPGA at power-up.

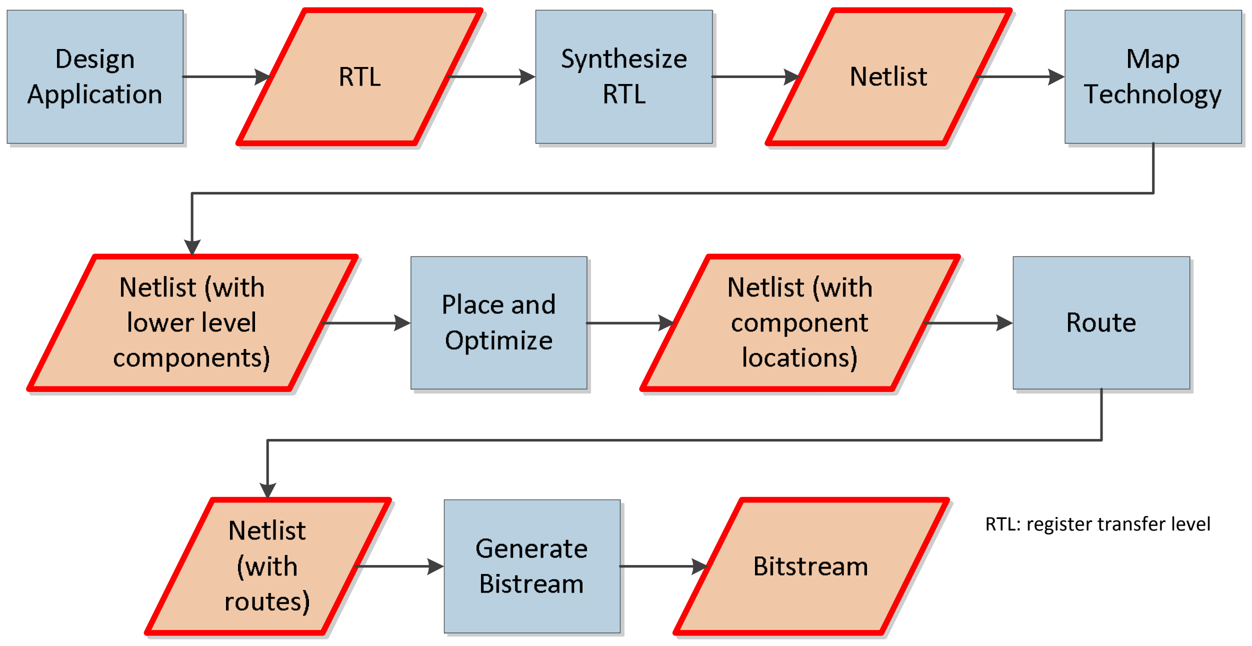


Figure 3. Application Development Flow

As shown in Figure 4, hardware development for an FPGA is abstracted into a simple flow that begins with identifying the context of the FPGA chip (that is, the supporting electronics and interfaces) in a bill of materials (BoM) and schematic. Included in this phase is the selection of the FPGA (and all other chips). The PCB is then designed and those design files, called Gerber files, are used to fabricate the PCB. The PCB is populated with all of the chips and discrete devices (such as resistors and capacitors) necessary, and the PCB with the unconfigured FPGA is ready for integration and use. Like in the application development flow, each of the artifacts could be compared against the requirements or previous artifacts in the flow for authentication, though with limited utility since most of the behavior is captured in the application development. For the most part, authentication methods in the hardware development phase are used to confirm that no additional functionality is included in either the PCB or the FPGA prior to integration.

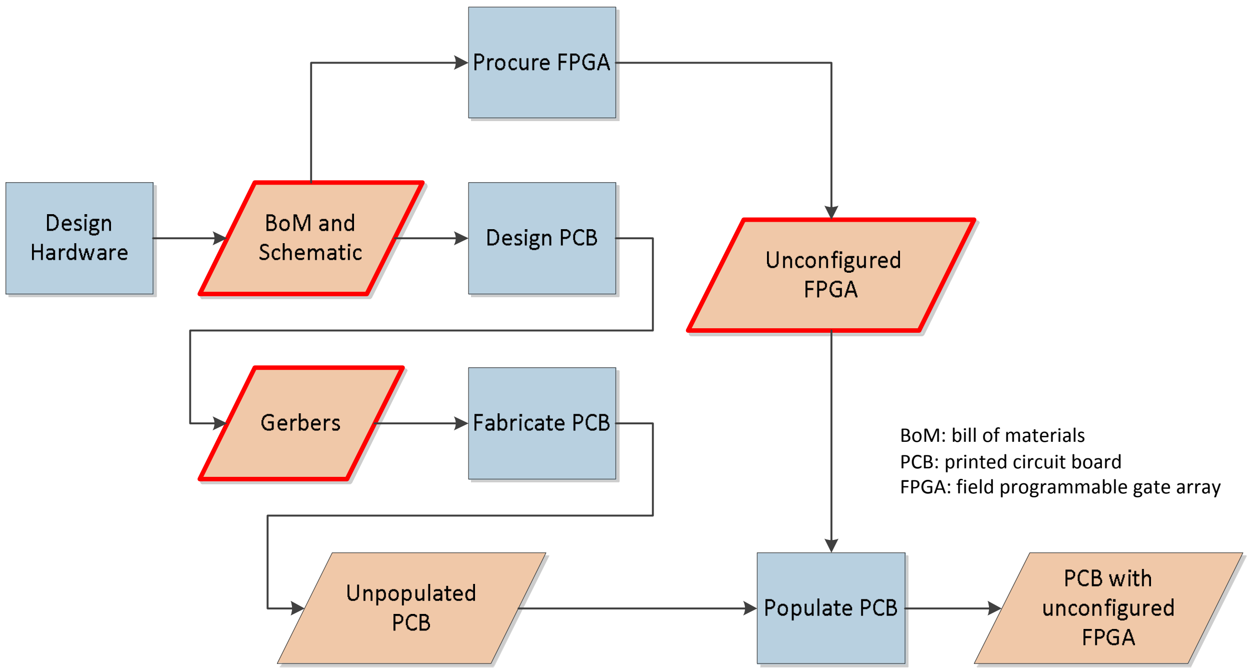


Figure 4. Hardware Development Flow

As shown in Figure 5, the system is prepared for operation, is operated, and is shut down at the end of its employment. The system could be used for authentication before, during, and after use. In addition, there are opportunities to authenticated the bitstream while it is being loaded onto the FPGA.

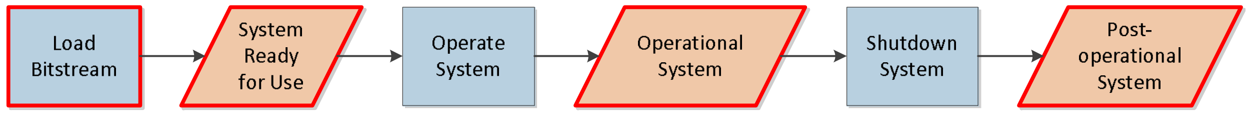


Figure 5. System Operation Flow

The authentication methods are organized around these three segments of the development and use lifecycle of an FPGA system: application development, hardware development, and system operation. In the first two segments, the authentication methods are intended to give the inspector confidence that the design (at various stages) meets the requirements. More specifically, these methods should give the inspector confidence that, at each point in the design progression, the design fulfills the functions agreed to in the requirements and includes no functionality that is not agreed to in the requirements. In the system operation segment, the inspector gains confidence that the system has not been altered at any point before or during use.

The list of authentication methods identified is shown in Table 1 and described in

FPGA Authentication Methods. In the table and within each appendix section describing the authentication methods, those that the Atomic Weapons Establishment (AWE) is investigating are listed first in orange, in priority order, followed by those that Sandia National Laboratories (SNL) is investigating in blue, in priority order. Each organization used a different method for assigning priority—AWE used a High/Medium/Low scale and SNL put them in rank order from highest priority (1) to lowest. The priority is used as a guideline to assist in planning the research effort only.

Table 1. Summary of Potential Authentication Methods

	Method	Lab	Priority	Appendix Section
Application Development	Bitstream authentication	AWE	High	**A.2.1**
	Formal methods with Solidify	AWE	High	**A.2.2**
	Constrained random testing with OSVVM	AWE	Medium	**A.2.3**
	Formal equivalence check (other than Onespin and Formality)	AWE	Medium	**A.2.4**
	Static analysis	AWE	Low	**A.2.5**
	Formal methods with Onespin	SNL	1	**A.2.6**
	Formal equivalence check with Onespin	SNL	2	**A.2.7**
	Formal equivalence check with Formality	SNL	3	**A.2.8**
	Constrained random testing with UVM	SNL	7	**A.2.9**
	Simulation test bench functional testing	SNL	8	**A.2.10**
	Code coverage simulation test bench	SNL	10	**A.2.11**
	Static analysis of source code	SNL	13	**A.2.12**
	Manual code inspection	SNL	15	**A.2.13**
Hardware Development	Depackaging and imaging (Flash)	AWE	High	**A.3.1**
	Depackaging and imaging (SRAM)	AWE	High	**A.3.2**
	Visual and x-ray examination of PCB	AWE	High	**A.3.3**
	Destructive PCB examination	AWE	Medium	**A.3.4**
	Depackaging and imaging (OTP/antifuse)	AWE	Low	**A.3.5**
	Depackaging and imaging (external PROM)	AWE	Low	**A.3.6**
	Manual hardware design verification	AWE	Low	**A.3.7**
	Scan chain verification (SRAM)	SNL	12	**A.3.8**
	Scan chain verification (Flash)	SNL	14	**A.3.9**
System Operation	Power analysis	AWE	Low	**A.4.1**
	Bitstream monitoring during configuration (SRAM)	SNL	4	**A.4.2**
	Bitstream comparison (SRAM)	SNL	5	**A.4.3**
	Bitstream comparison (Flash)	SNL	6	**A.4.4**
	Bitstream real time monitor with built-in CRC or hash	SNL	9	**A.4.5**
	Functional test of hardware with real or simulated inputs	SNL	11	**A.4.6**

3. FPGA Basis System

An FPGA basis system was designed and built in order to support evaluation of the prioritized authentication mechanisms on the same target design and system. In addition, it was desired for the system to fulfill an application that was relevant to our investigation—that is, an application that could be used in treaty monitoring or verification. The system to be evaluated is a portal monitor that both detects the presence of radioactive materials moving past it and determines the direction of motion. The system will include an FPGA for processing and will have simulated detectors (though it is designed so that it could be evaluated with real detectors).

3.1 Monitoring Scenario

Two detectors are installed on a wall in a hallway with a processor module between them. Various objects move through the hallway in both directions. The functions of the portal monitor are to detect and indicate the detection of 500 g (or more) of weapons grade Pu (90-100% ratio of ^{239}Pu to ^{240}Pu, with aged Pu), shielded with ¼" of lead, as it moves down the hallway, and indicate the motion of direction. All geometries are shown in Figure 6 and are described in Table 2. The Pu source moves between 0.1 and 2 m/s.

Table 2. Dimensions in Monitoring Scenario Geometry

Dimension	Description	Value or Range
d_{dz}	Distance from the edge of the detection zone to the edge of the detector	2 m (arbitrary)
w_d	Width of each detector	15.2 cm
o_d	Offset between the detectors	1 m (arbitrary)
d_m	Lateral distance between the source and the front face of the detectors when the source is directly in front of the detector	0.2 – 5 m (arbitrary)

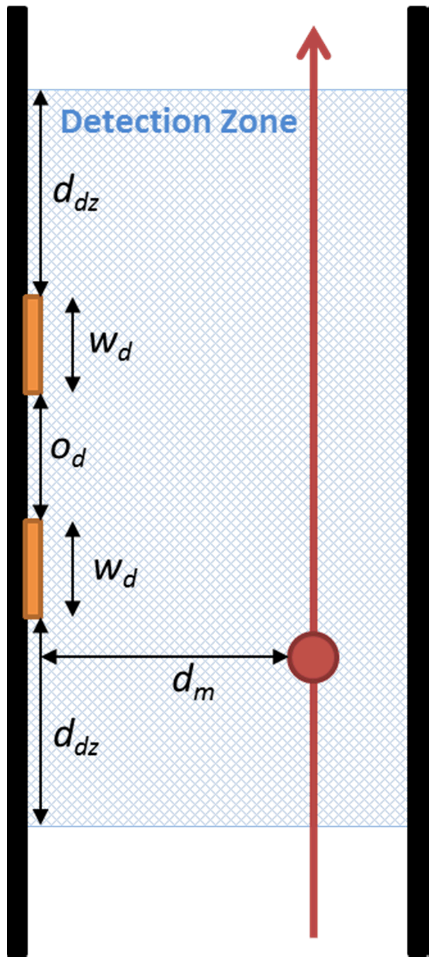


Figure 6. Geometry of Portal Monitoring Scenario

The detector responses with the source at each point in the detection zone are shown in Figure 7 and Figure 8, with a lateral distance (d_m) of 25 cm and 300 cm, respectively. The detector responses were created in MCNP[2] with an environment consisting of steel-reinforced concrete floors and walls. The detector response is presented as counts per second at each position with background subtracted.

[2] MCNP is Monte Carlo N-Particle, a tool developed by Los Alamos National Laboratory for modeling transport of neutrons, photos, electrons, or coupled particles.

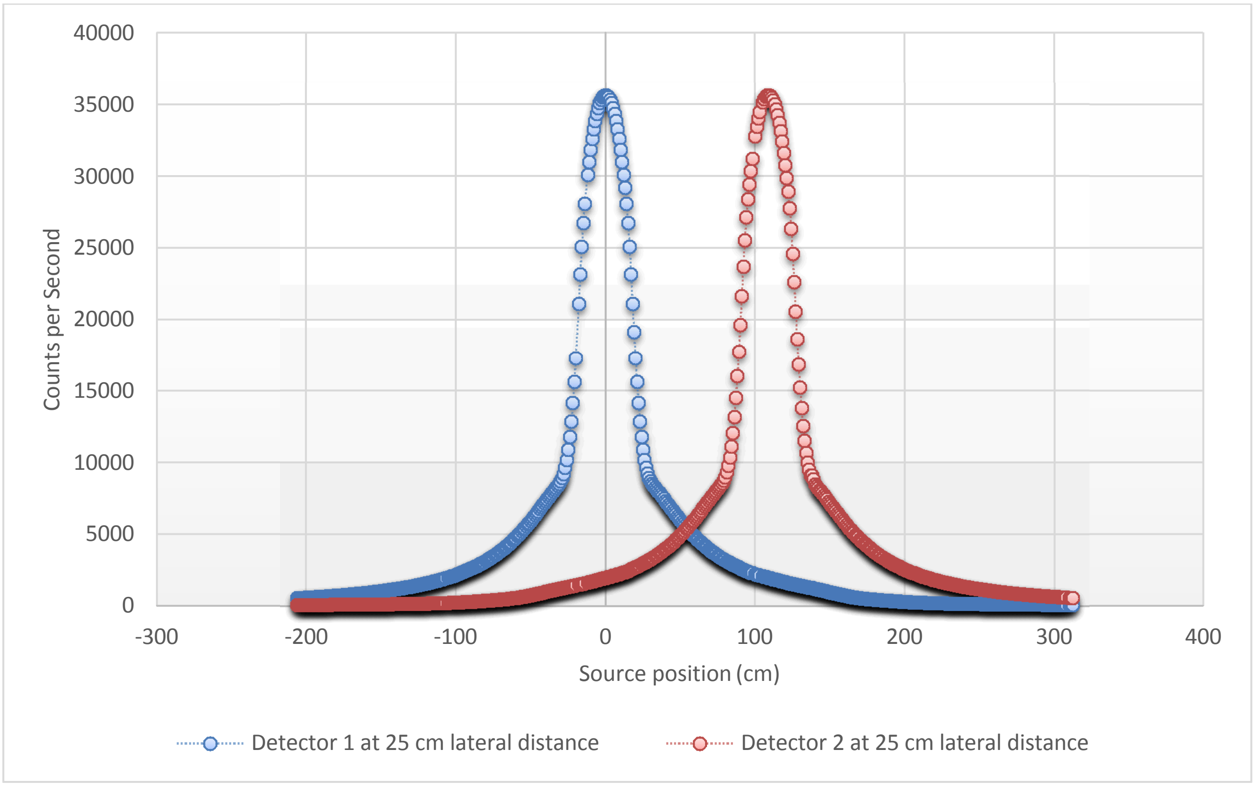


Figure 7. Simulated Counts on Each Detector with Source Moving at 25 cm from Detector Face (d_m = 25 cm)

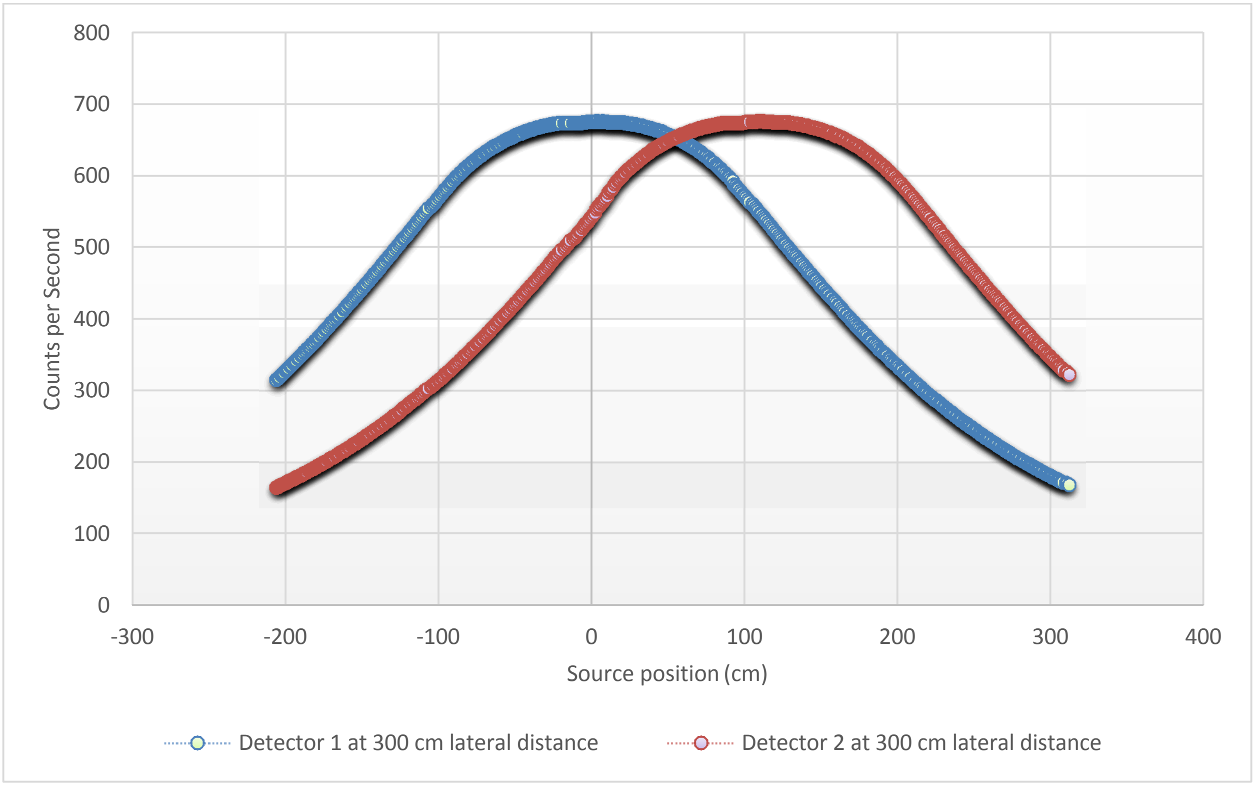


Figure 8. Simulated Counts on Each Detector with Source Moving at 300 cm from Detector Face (d_m = 300 cm)

3.2 FPGA Basis System Functional Design

The portal monitor system consists of two detectors (a left detector and a right detector) with a processor module (the FPGA basis system) in the middle. This functional design describes the inputs, outputs, and behavior of the FPGA basis system. The logical design of the resulting basis system is described in

The inputs of the FPGA basis system are:

1. LD: Left detector pulses (15 µsec in length and 3.3 V in height for each detected gamma ray)
2. RD: Right detector pulses (15 µsec in length and 3.3 V in height for each detected gamma ray)
3. LR: Latch reset button (or other momentary actuator)

The outputs of the FPGA basis system are:

1. LRA: Left-to-right alarm indicator (an LED)
2. RLA: Right-to-left alarm indicator (an LED)
3. HBA: High background alarm indicator (an LED)
4. LBA: Low background alarm indicator (an LED)
5. R: Ready indicator (an LED)

The behaviors of the FPGA system are:

1. Collect counts from each detector for each 100 ms using 10-bit non-overflowing counters
   a. LD and RD are defined here as the count each 100 ms
2. Calculate a moving average (MA) for each detector for the last 204.8 s
   a. MA(LD) and MA(RD) are the average counts per 100 ms over the last 204.8 s
   b. 204.8 seconds is used because it is equal to 2048, or 2^{11}, time slices of 100 ms
3. If LD > n*sqrt[MA(LD)] + MA(LD) and then RD > n*sqrt[MA(RD)] + MA(RD), assert LRA
   a. n is a settable threshold that defaults to 4.0
4. If RD > n*sqrt[MA(RD)] + MA(RD) and then LD > n*sqrt[MA(LD)] + MA(LD), assert RLA
   a. n defaults to 4.0
5. If MA(LD) > 300 or MA(RD) > 300, assert HBA
6. If MA(LD) < 50 or MA(LD) < 50, assert LBA
7. If LR is pushed, remove all asserted indicators
8. On power up, assert all alarms
9. On power up, assert R (solid light) and disable all detection functionality (behaviors 2-6) for the first 204.8 s
   a. After the first 204.8 s, blink R and enable all detection functionality (begin behaviors 2-6)

3.3 FPGA Basis System Physical Design

Two physical designs were fabricated to evaluate the same functional design for authentication – one featuring a Microsemi FPGA and one featuring a Xilinx FPGA. These two chips have significant differences that may lead to different authentication considerations. Each design consists of a printed circuit board (PCB) with an FPGA, associated components, interfaces for the detector inputs, a switch for the latch reset, and LEDs for the outputs. The complete design specification for each printed circuit board, which includes a circuit schematic and images of each layer of the printed circuit board layout, are included in the files "PM_Microsemi.pdf" and "PM_Xilinx.pdf". Shown here are the PCB layout images showing all layers for the Microsemi design (Figure 9) and the Xilinx design (Figure 10). The bill of materials for each board is in Table 8 and Table 9 (in Appendix D) for the Microsemi design and the Xilinx design, respectively. The built PCB is shows in Figure 9 and Figure 10 for the Microsemi design and the Xilinx design, respectively.

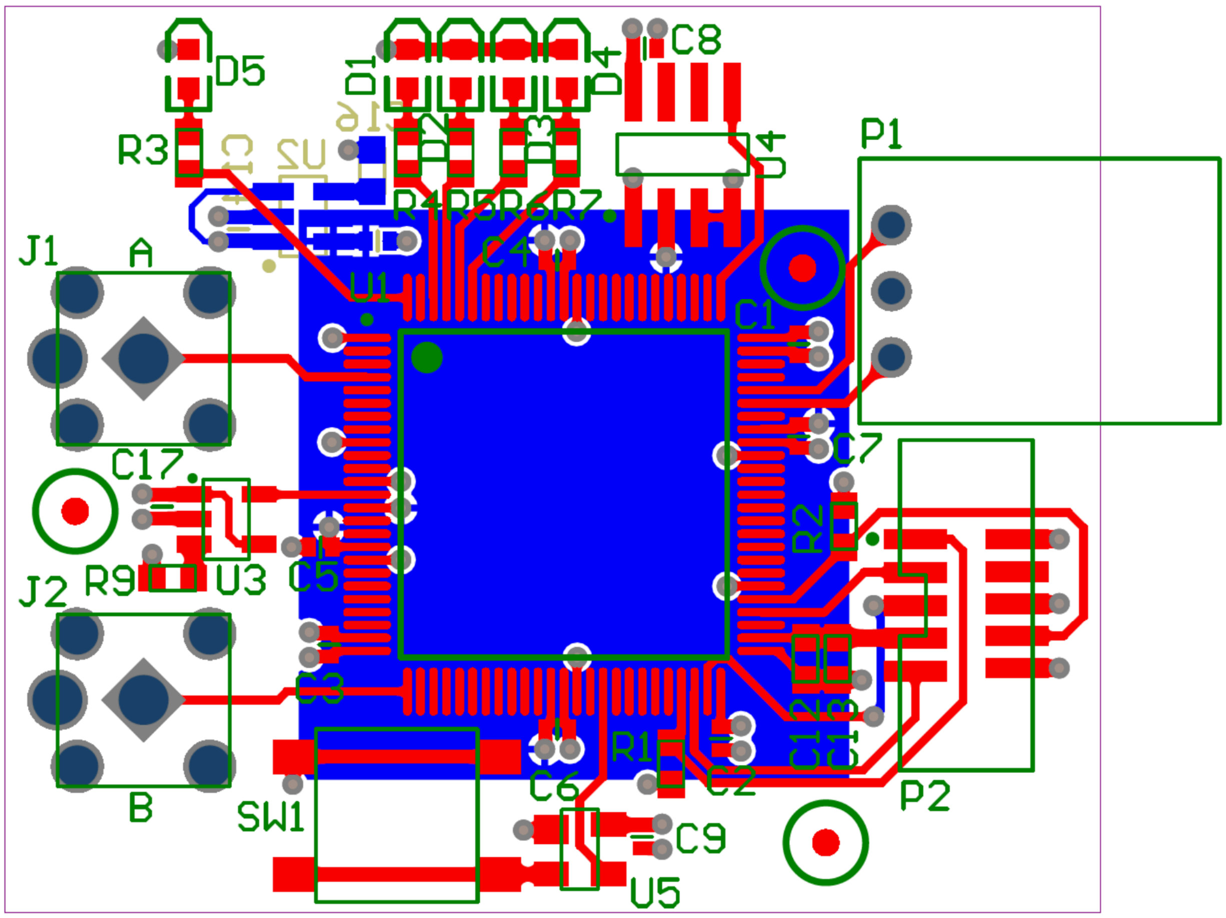


Figure 9. Basis System PCB Layout with Microsemi FPGA

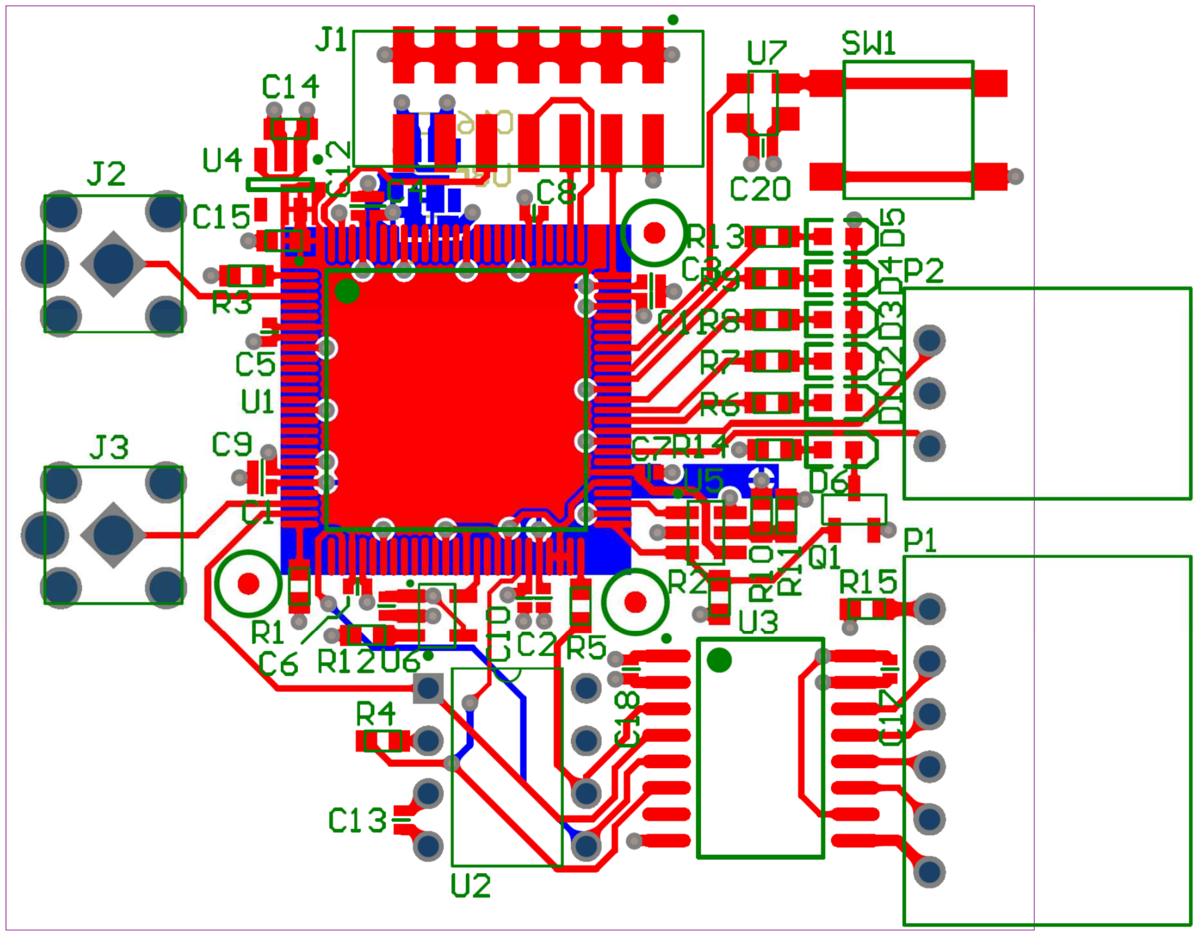


Figure 10. Basis System PCB Layout with Xilinx FPGA

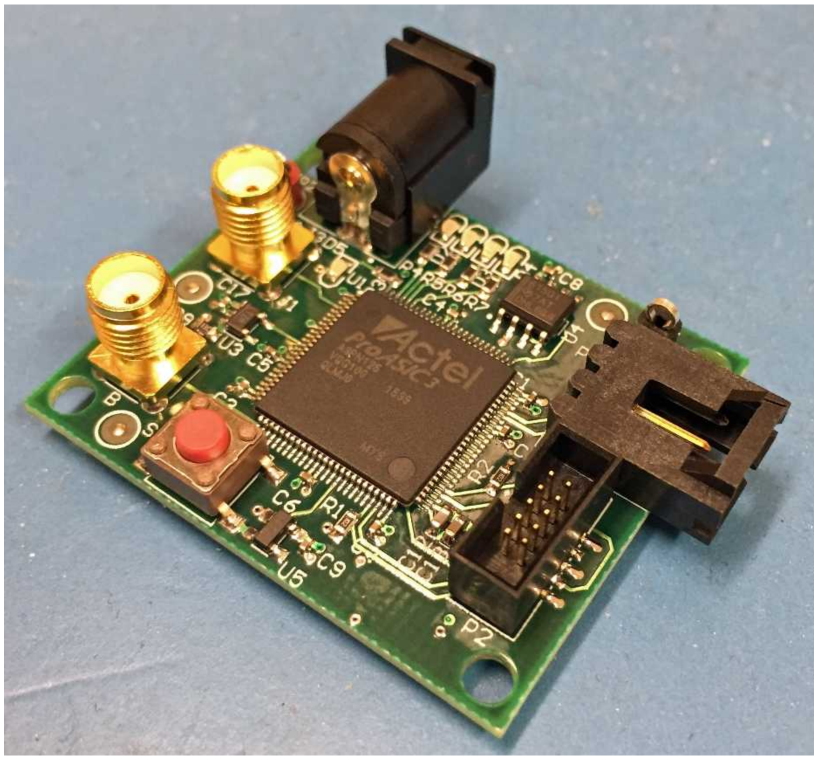


Figure 11. Built Basis System PCB with Microsemi FPGA

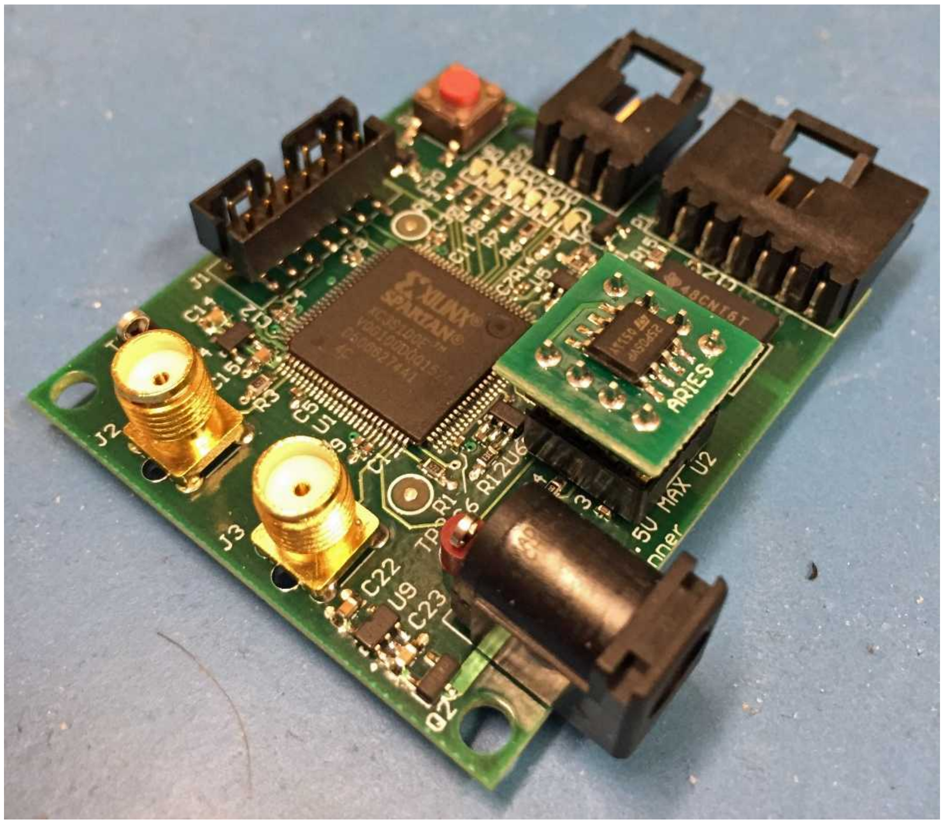


Figure 12. Built Basis System PCB with Xilinx FPGA

3.3.1 FPGA Choice

The two main drivers for our choice of FPGA in this project were:

- Available in non-ball grid array (BGA) package. This package necessitates having traces hidden underneath the part itself. Alternative packages, such as quad flat pack (QFP), have externally visible leads and are easier to visually inspect.
- Large feature size. To facilitate optical imaging, we chose the FPGA with the largest possible feature size. This choice limits us to older generations of FPGAs, running the risk of part deprecation. However, these older FPGAs also contain less logic and features then newer FPGAs, reducing the opportunity for design subversion in the unused areas of the device.

Beyond these two requirements, the only other driver was that the design fit in the selected device. Thus, we choose the smallest possible device that could contain the entire design. For this particular application, the limiting factor was the availability of block random access memory (RAM). Block RAM is a built-in memory within a Xilinx FPGA. In our design, we used this memory to store the sampled count rates as we built up a moving average. The size of the memory needed was 11 bits for address and 10 bits for data, effectively 20 kb. However, block RAMs can only be configured with even-sized data widths in multiples of two (×2, ×4, ×8, ×16), so our effective memory needs were actually 2k×16, or 32 kB. Luckily, even the smallest available Xilinx FPGA has enough memory bits to meet our needs.

Given the above criteria, two Xilinx FPGAs met our needs, the Spartan-3 XC3S50 and the Spartan-3E XC3S100E. However, because we also wanted to experiment with abandoning the internal memory and storing our data in internal registers instead, the XC3S50 was not a viable choice as it did not contain enough internal registers to meet this need. Thus we chose the XC3S100E. The final resource utilization statistics for the implemented design are:

- 22% usage of device logic (slices)
- 100% usage of device memory (ramb16s)
- 15% usage of device inputs/outputs (I/O)

4. Evaluations of Authentication Methods

Several of the identified authentication methods were evaluated for their feasibility, performance, and costs on the FPGA basis systems, after they were built and tested. This section describes the results of those evaluations.

4.1 Evaluation Approach

In order to evaluate each authentication method in a similar fashion, evaluation criteria have been developed. The results of the evaluation of each method will be reported in terms of these criteria:

- **Performance**
  - *Feasibility*: does this method perform the function that it is intended to perform?
  - *Result quality*: how valuable is the result of this method in verifying trust in the equipment?
    - Coverage: can we find all of the bugs? If not, what percentage can we find?
    - Probability of finding a bug given that there is a bug (true positive rate)

        - Probability of not finding a bug given that there are no bugs (true negative rate)
    - *Refutability*: is the evidence collected unambiguous?
- **Cost**
    - *Personnel*
        - Number of people hours needed to perform the method
        - Specific expertise needed to perform the method
    - *Cost of tools*
        - Computing resources needed to perform the method
    - *Time*

Since there are many unknown aspects about how these methods would be applied to a design in the future, most of the evaluations deal with the performance of the method and the costs are fairly simple estimations.

To support the performance criteria evaluation, we have created five "buggy" versions of the same VHDL design, as described below in Section 4.1.1: Summary of Modified Designs . The evaluators leading the formal methods and formal equivalence verification techniques were not aware of the modifications and were asked to not only identify them, but explain their nature. While the probability of not finding a bug given that there are no bugs will be reported as a single data point (i.e., does the method function correctly on the single "clean" version of VHDL?), the probability of finding a bug given that there is a bug will be tested with the five "buggy" versions of varying cleverness of bugs. If these evaluations prove to be useful, we intend to expand this in the future to support the development of a true positive rate and false positive rate with more precision.

The other criteria will be judged by expert opinion. Refutability may be hard to assess, but we will make some judgement about whether the evidence collected could be explained any other way. The cost criteria will be based on the expert opinion of the engineers and scientists using the tools for these evaluations.

4.1.1 Summary of Modified Designs

In addition to the unmodified reference design, we created five additional designs, each containing a modification to the base VHDL code that subverts the operation of the portal monitor in a way that could potentially lead to undeclared activities not being detected. While we could have more cleverly inserted the malicious code to be harder to locate with manual code inspection, our focus was instead on evaluating the authentication techniques irrespective of the degree of difficulty of malicious logic insertion.

Modified Design #1

This subversion targets the logic that processes the square root of the background, the result of which is then used to set the alarm thresholds. The modified file is sqrt.vhd. The source VHDL originally contained:

```
root <= root_r;
```

This line of code simply assigns the result of the square root operation to the output of the module. The subverted design modifies that line as follows:

```
root <= root_r + (root_r'RANGE => '1')/2;
```

This new code arbitrarily increases the result of the square root operation by 1.5x. For small values of square root, the new value still provides acceptable material alarm levels. However, for increasingly large backgrounds, the alarm thresholds may exceed detectable values. By raising the background to a high enough level (but lower than the high background alarm level), the host may be able to sneak material in or out of the facility.

Modified Design #2

This subversion targets the wires that connect the background alarms to the top level I/O of the FPGA. The modified file is detector.vhd. The original source code directly connects these signals to each other:

```
High_background_alarm => High_background_alarm,
Low_background_alarm  => Low_background_alarm
```

The subverted design disconnects these wires and permanently disabls the background alarm outputs:

```
High_background_alarm => open,
Low_background_alarm  => open
);
High_background_alarm <= '0';
Low_background_alarm  <= '0';
```

The high and low background alarms will never activate in the subverted design. While this change is easily detectable with functional testing, it provides an interesting corner case as no actual logic has been altered, but instead we have rewired the outputs of circuits.

Modified Design #3

Besides logic and wiring changes, another subversion tactic is to change the value of constants used throughout the design. This subversion alters the values of the constants used to set the high and low background levels in a way that prevents both alarms from ever activating. However, it attempts to do so in a moderately clever way, by carefully altering the spelling of both constants to still look correct. Such a slight alteration may not be diagnosable during a manual code inspection. The modified files are constants.vhd and alarm processor.vhd. Originally, the VHDL contained the following code (from constants.vhd):

```
constant HIGH_ALARM_LEVEL : natural := 10#30#;
constant LOW_ALARM_LEVEL  : natural := 10#5#;
```

and (from alarm_processor.vhd):

```
if unsigned(Average) >= HIGH_ALARM_LEVEL then

if unsigned(Average) <= LOW_ALARM_LEVEL then
```

The subverted VHDL appears as follows (from constants.vhd):

```
constant HI_ALARM_LEVEL : natural := 10#1022#;
constant LO_ALARM_LEVEL : natural := 10#1#;
```

and (from alarm_processor.vhd):

```
if unsigned(Average) >= HI_ALARM_LEVEL then

if unsigned(Average) <= LO_ALARM_LEVEL then
```

This modification would also be detectable through functional testing, but tests another interesting subversion avenue.

Modified Design #4

This subversion is another logic change. It modifies the logic that counts incoming pulses from the radiation detectors. The modified file is pulse_counter.vhd. The original VHDL simply counts every rising edge of a pulse as long as the counter has not saturated:

```
        count_r <= count_r + 1;
```

The subversion alters this counting to ignore every other pulse once the counter has reached ¼ of its maximum value:

```
        if count_r > (count_r'RANGE => '1')/4 then
                skip_count <= not skip_count;
                if skip_count = '0' then -- skip every other count
                        count_r <= count_r + 1;
                end if;
        else
                count_r <= count_r + 1;
        end if;
```

Such a change artificially suppresses the count rates of high activity sources but would not be noticeable during functional testing so long as the check source used produced lower count rates than the actual material.

Modified Design #5

For the final subversion, an attempt was made to hide the modification as cleverly as possible. This change alters the top_level.vhd file and alters how the two sides of the detectors are combined to produce a single alarm. The original code VHDL is:

```
High_background_alarm <= high_background_alarm_A or high_background_alarm_B;
Low_background_alarm  <= low_background_alarm_A or low_background_alarm_B;
```

In the subverted design:

```
High_background_alarm <= high_background_alarm_A xor high_background_alarm_B;
Low_background_alarm  <= low_background_alarm_A xor low_background_alarm_B;
```

In the original design, if either side of the portal monitor detects a high or low background, the corresponding alarm is automatically set (via the OR gate). However, in the subverted design, the alarms are set if only one side or the other is alarmed. If both sides alarm, the alarm is not activated. This alteration might not be noticed during functional testing if a clever host puts a source or shielding near only one side of the portal to activate the alarm but does so to both sides simultaneously during actual operation.

4.1.2 Evaluated Methods

The choice of methods to evaluate was based on the prioritization, the effort needed for each evaluation (some methods were evaluated on the basis system, while others were only surveys of applicable tools and techniques, and therefore had a lower research cost), and the resources available to each team. The methods evaluated are summarized in Table 3 below.

Table 3. Evaluated Authentication Methods

	Method	Lab	Type	Section
Application Development	Bitstream authentication	AWE	Survey	4.2
	Formal methods with Solidify	AWE	Application	4.3
	Formal methods with Onespin	SNL	Application	4.4
	Formal equivalence survey	AWE	Survey	4.5
	Formal equivalence check with Onespin	SNL	Application	4.6
	Formal equivalence check with Formality	SNL	Survey	4.7
Hardware Developme	Physical FPGA authentication	AWE	Survey	4.8
System Operation	Bitstream monitoring during configuration (SRAM)	SNL	Application	4.9
	Bitstream comparison (SRAM)	SNL	Application	4.10
	Bitstream comparison (Flash)	SNL	Survey	4.11

4.2 Bitstream Authentication Survey

Bitstream authentication is the process of proving the authenticity and integrity of the sequence of bits (or bitstream) used to program or configure an FPGA. A survey of existing research on bitstream authentication returns two papers[3,4] that give a good indication of the ways that bitstream authentication is perceived in a commercial sense.

Not surprisingly, the commercial sector is concerned with theft and cloning of intellectual property (IP). An obvious form of attack is passive eavesdropping of the bitstream during its transmission to an FPGA. However, this should be of no concern in a treaty verification context because there is typically no secrecy in the design. Preventing the active manipulation of a bitstream during its transmission to an FPGA, on the other hand, is of concern in treaty verification and this is the principal purpose of bitstream authentication in this context.

In the first of these papers, Parelkar correctly points out that bitstream encryption, which prevents IP theft, is not adequate for bitstream manipulation because it is possible to damage the FPGA with a manipulated ciphertext even if the decrypted bitstream is ‘gibberish’. The paper proposes various authentication techniques to prevent manipulation of the bitstream, but these are just variations of a single approach in which the bitstream is hashed and the hash digest is included as part of the transmitted bitstream. On receiving the bitstream the FPGA can perform its own hashing function to determine whether the hash values match and, therefore, it has a way of checking whether the bitstream is authentic. Secure hashing (involving a secret ‘key’ of some form) is necessary to deduce that the bitstream came from an authentic source; otherwise the originator of the hashed bitstream cannot be confirmed and bitstream authentication cannot be achieved.

[3] M. Parelkar: FPGA Security – Bitstream Authentication, George Mason University
[4] S. Drimer: Authentication of FPGA Bitstreams – Why and How, University of Cambridge

The paper is focused on the implementation details of bitstream authentication by the FPGA itself. As such, the logistical issues of FPGA bitstream authentication are largely ignored. Assuming that the FPGA is protected by a tamper resistant or tamper evident casing (to prevent or detect a direct attack on the FPGA itself) several important issues are not addressed:

- No consideration is given to key distribution in the hash message authentication code (HMAC) approach and what role, if any, public key cryptography plays in this process. This is a nontrivial problem on which the security of the system rests.
- The design of the pre-processor in the implementation of Secure Hash Algorithm 1 (SHA-1) expects the first word to be 'the length of the message to be processed'. It is not clear whether this a potential weakness of the method which could be exploited to undermine the security of the system. (This question is, of course, in addition to existing perceived weaknesses of SHA-1 itself.[5])
- The procedure for authenticating HMAC bitstreams is not given in any detail.
- How is the code for the hashing algorithm itself authenticated and deployed on the FPGA?

In the second paper, Drimer also begins by describing the inadequacies of encrypting bitstreams for authentication. However, they do not qualify the amount of damage that a manipulated ciphertext could cause to an unsuspecting FPGA, since it relies on other factors (such as high currents in supporting circuitry) to damage the FPGA physically. Nevertheless, encryption and other methods such as a cyclic redundancy check (CRC) does not provide the necessary security protection to prevent the deployment of unauthorised bitstreams. Error detection and correction codes such as CRCs prevent (accidentally) corrupted bitstreams from being loaded onto an FPGA. They do not detect all types of error, and they can be forged with relative ease so that a manipulated bitstream will pass a CRC.

The scenario described in Section 2.2 of Drimer has an element of secrecy because the key for the HMAC is embedded within the FPGA prior to use. This is likely to be unacceptable in a treaty verification setting because there will be a feature on the FPGA that is unknown to all parties. It is also questionable whether this approach is adequate when authentication of the hardware itself is likely to be required.

The role of authenticated encryption techniques, as discussed by both authors of these papers, seems to be an attempt to appease FPGA manufacturers who do not see worthwhile value in providing separate authentication functionality in addition to pre-existing encryption mechanisms for protecting IP. Whilst using the same circuit for encryption and authentication is desirable from a manufacturer's perspective, authenticated encryption provides no additional benefit to the authentication of bitstreams from a user perspective; in a treaty verification context, the secrecy provided by the encryption aspect is superfluous. Bitstream authentication for treaty verification remains a difficult challenge.

4.3 Formal Methods with Solidify

Averant's Solidify tool[6] is a model checker that specifically targets hardware description language (HDL) models. In the context of this work the models are written in VHDL. The tool includes a VHDL compiler which translates synthesisable VHDL into an intermediate format for use in Solidify's static verification engine. In this respect Solidify is an *explicit state* model checker rather than a symbolic model checker. This means that in order to explore the entire state space of the model Solidify has to check all possible values that could be assigned to the variables and signals contained within the model. (This is in contrast to symbolic model checking in which the analysis is performed on variables and signals that are

[5] See, for example: https://www.schneier.com/blog/archives/2015/10/sha-1_freestart.html
[6] http://www.averant.com/products-solidify.html

represented symbolically, i.e., without assigning values explicitly.) The advantage of explicit state model checking is that the analysis mimics the actual execution of the code. The disadvantage is that analysis of models with a large state space are susceptible to state space explosion, making the tool infeasible.

4.3.1 Theory of Operation

Properties are specified using Averant's Hardware Property Language (HPL). Solidify accepts properties written in more familiar property languages such as Property Specification Language (PSL), SystemVerilog Assertions (SVA), Open Verification Library (OVL), and Openvera Assertions (OVA), but these are translated into HPL prior to analysis. Hence, HPL is more general than these languages and contains an extensive range of operators for property specification.

There is conflict, however, between the expressiveness of the property language and the extent to which an analysis can run to completion (i.e., without running out of memory or exceeding acceptable time limits); properties that describe complex behaviour over many time steps are likely to fail to complete. The tactic employed in this investigative work has been to perform analysis on individual modules with simple properties that are restricted to the states of the current clock cycle and only one or two clock cycles into the future. This is not as constraining as it might seem because, as the tool documentation encourages, users of the tool should 'think inductively'. This amounts to specifying and analysing initialisation properties (i.e., properties of all the 'initial' states), and then specifying and analysing properties of all possible 'next' states that are reachable from a constrained set of 'current' states. For a time-sensitive VHDL architecture, this implies two kinds of property checks:

- What happens to the state on (asynchronous) reset?
- What happens in the 'next' clock cycle given the state in the current clock cycle?

Such properties are usually short and require an analysis that is within the capabilities of the Solidify model checker. An example of the first kind is:

```
Reset => current_state == 0;
```

(Here => indicates logical implication, such as an if/then statement, and == represents equality.) A successful analysis of this property demonstrates that an asynchronous reset always results in a state in which the signal `current_state` is assigned a value 0. An example of the second kind of property is:

```
!Reset && `X(!Reset) && clk_edge && load => `X(b) == h;
```

(Here `!` represents logical negation, `&&` indicates logical conjunction, and `` `X `` represents the 'next' clock step.) In this case, the property states that, on the assumption that there are no resets in the current time step and the next time step, and the signals `clk_edge` and `load` are asserted in the current state, the signal `b` is the value `h` in the next time step.

Even though the second of these two examples is phrased in terms of the current and next states, the success of its analysis using Solidify informs us that it is an *invariant* property. This means it is true in all possible states. As a consequence, it is possible to infer other properties without the need for model checking. In the case of the example above, it is also possible to infer from the previous result:

```
`X(!Reset) && `X`X(!Reset) && `X(clk_edge) && `X(load) =>
   `X`X(b) == h;
```

without having to resort to further model checking. This inferred property gives the value of `b` in two time steps (with respect to the modified antecedent). Due to its invariance, the original property can be used to infer similar variations for any number of time steps.

To infer more complex, longer term emergent behaviour of the VHDL design requires yet further analysis of the short-term properties obtained from Solidify model checking. In general, such behaviour can be inferred formally by taking the logical conjunction of multiple properties (to reason about the interaction of different parts of the VHDL design) over multiple time steps to extract longer term properties, all of which is done externally to the Solidify tool. This general approach has been reasonably successful in analysing emergent properties of simple VHDL models. There is currently no favoured or specific methodology; this is the subject of ongoing research.

In addition to the example given above, some other ways to derive new properties from old include:

- Strengthening of the antecedent: from a property of the form `A => C` infer, for any `B`, the property `A && B => C`;
- Conjunction of consequents: from two properties of the form `A => B` and `A => C`, infer the property `A => B && C`;
- Transitivity of implication: from two properties of the form `A => B` and `B => C`, infer the property `A => C`;
- Modus ponens, i.e., from the two properties `A` and `A => B`, infer the property `B`.

As an example, consider the following three properties of a VHDL model that are confirmed by model checking:

```
!Reset && `X(!Reset) && clk_edge && current_state == 0 && !start =>
‘X(current_state) == 1;
```

```
!Reset && `X(!Reset) && clk_edge && current_state == 1 &&  start =>
‘X(current_state) == 2;
```

```
!Reset && `X(!Reset) && clk_edge && current_state == 2 =>
‘X(current_state) == 3;
```

The second of these can be rewritten in terms of the next time step due to its invariance:

```
`X(!Reset) &&`X`X(!Reset) && `X(clk_edge) && ‘X(current_state) == 1 &&
‘X(start) => ‘X(‘X(current_state)) == 2;
```

Transitivity of this derived property with the first of the properties given above yields:

```
!Reset && `X(!Reset) &&`X`X(!Reset) && clk_edge && `X(clk_edge) &&
current_state == 0 && ‘X(current_state) == 1 && !start && ‘X(start) =>
‘X(‘X(current_state)) == 2;
```

This reveals the value of `current_state` after two time steps (assuming the antecedent is true) without model checking.

4.3.2 Performance Results

In a similar manner to the OneSpin approach, each of the VHDL modules has an associated set of properties that are checked by the Solidify tool. For the purposes of this work the properties are expressed in Solidify’s native property language, HPL. Each property describes the intended behaviour of its corresponding VHDL module over one or two time steps in accordance with the theory of operation described above. On termination (and without any out-of-memory errors) the model checker reports whether properties have passed or failed. A pass means that the property holds at every time step in all possible executions of the VHDL module (i.e., it is an invariant property). A fail means that there exists at least one instance of the possible executions during which the property fails. A fail is accompanied by a counterexample which describes (as a wave diagram by default) the failure in more detail.

Feasibility

In order to the set up the model checker to perform the Solidify analysis, the reset sequence and the values of any generics must be declared in advance. A number of general ‘autochecks’ can be performed on the VHDL design prior to the analysis of the application-specific properties. These include a boundary check, a deadlock check, a deadcode check and a reset check. The reset check can be used to verify that the reset sequence declared as part of the setup actually resets the VHDL modules.

As stated above, each property is associated with an individual VHDL module. However, by analysing each module individually, the interactive behaviour between modules is lost. Consequently, an individual module which is constrained by the interaction with its neighbouring modules is analysed with no such constraints which could result in false negatives. It could be necessary, therefore, to model the interaction between modules in order to provide a more accurate environment for the analysis. This can be done in Solidify by adding assumptions to the property file. For example, if a property `a > 0` has been proven in module that outputs values on signal `a` and interacts by inputting the value of `a` to another module then the property set of this latter module can assume `a > 0` (without further proof). Therefore, any analysis of this module does not have to consider states in which, for example, `a = 0`. This approach has the additional benefit of reducing the state space that the model checker has to explore in order to verify a given set of properties.

Evaluation of Original Design

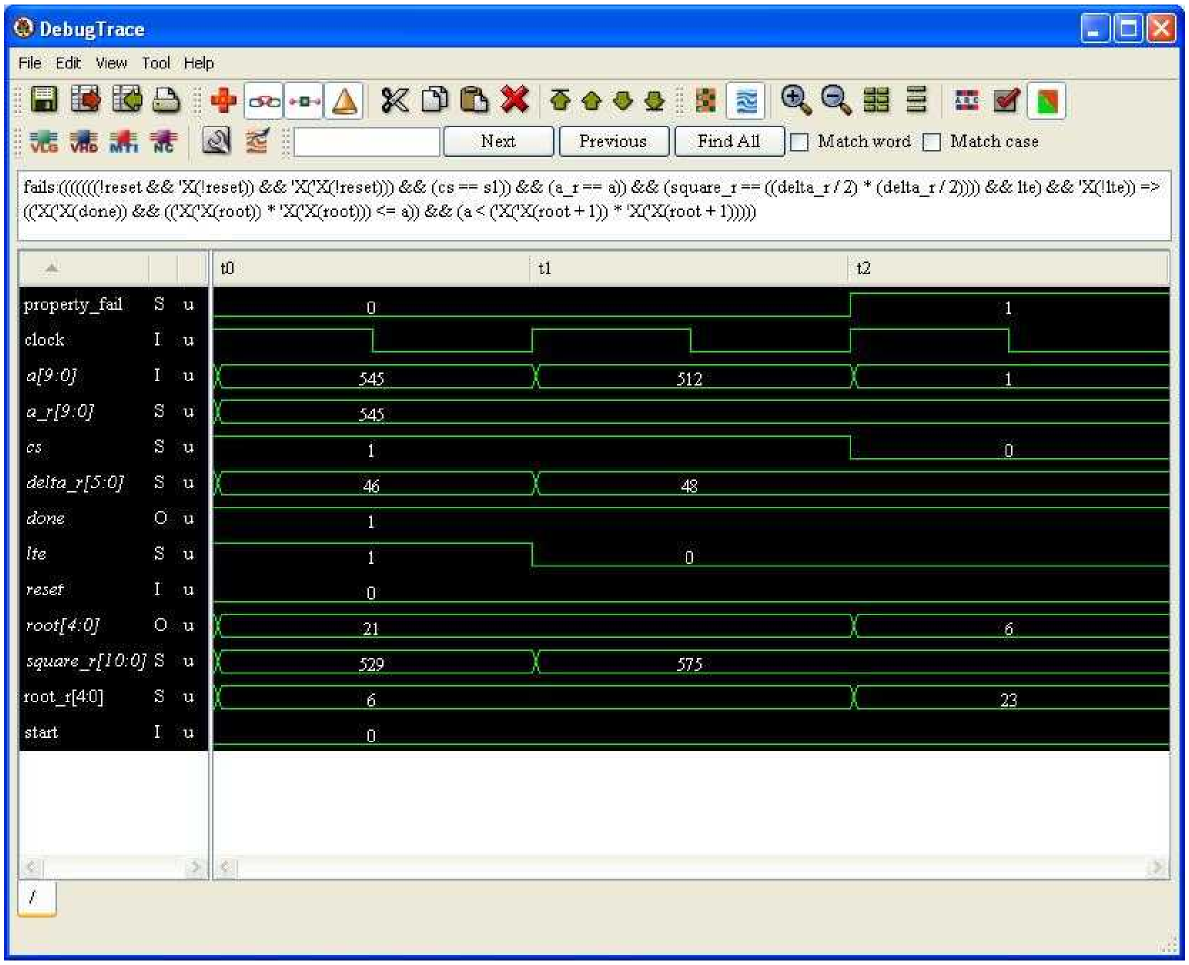
The properties derived for the purposes of this demonstration were obtained from the low-level description of the VHDL code and the code itself; the consequence of this is that model checking of the properties passes in all cases. As stated above, this could be detrimental to the independence of the properties with respect to the design. Ideally the properties are derived from a higher-level/implementation independent description of the application, but a certain amount of low-level detail is necessary in order to refer to components of the VHDL design (but also to restrict the analysis of properties to one or two future execution time steps).

Evaluation of Modified Designs

The five subversions were submitted individually for Solidify analysis. Repeating the analysis of the same properties that were used for the unmodified code results in failure with counterexamples in all cases, thus demonstrating that the Solidify model checker is capable of detecting such modifications to the VHDL design. This is not altogether surprising because the properties specified for the unmodified code are in close correspondence with the code itself. The following screen shots demonstrate some of the actual counterexamples and show how Solidify displays its results.

Modified Design #1

The modification to the assignment of the signal `root` in `sqrt.vhd` is revealed by repeating the Solidify analysis, as the following counterexample demonstrates



```
fails:(((((((!reset && 'X(!reset)) && 'X('X(!reset))) && (cs == s1)) && (a_r == a)) && (square_r == ((delta_r / 2) * (delta_r / 2)))) && lte) && 'X(!lte)) =>
(('X('X(done)) && (('X('X(root)) * 'X('X(root))) <= a)) && (a < ('X('X(root + 1)) * 'X('X(root + 1)))))
```

Figure 13. Modified Design #1 Counterexample Trace

The failing property (i.e., the property in the debug trace window) shows that it is not the case that

`‘X(‘X(root))*‘X(‘X(root)) <= a && a < ‘X(‘X(root+1))*‘X(‘X(root+1))`

which states that `root` is the (integer) square root of `a`. The signals of the trace show exactly how this property is violated in the counterexample. This occurs specifically at time step `t2` (as indicated by the `property_fail` trace).

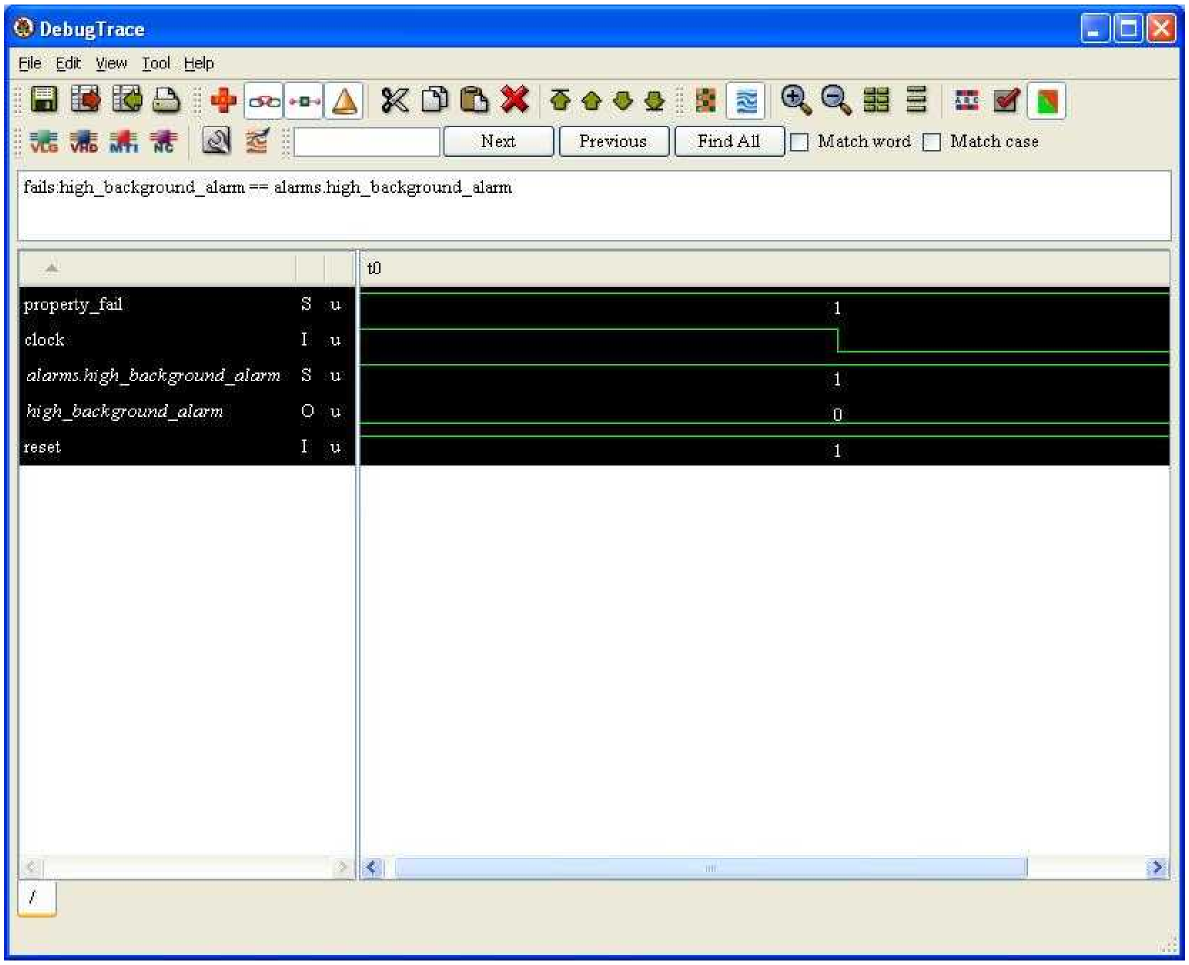
Modified Design #2

The modification to the High_background_alarm and Low_background_alarm in the alarms entity of detector.vhd is revealed by Solidify model checking. The properties referring to these signals are as follows

```
High_background_alarm == alarms.High_background_alarm
```

```
Low_background_alarm == alarms.Low_background_alarm
```

The dotted prefixes `alarms` are used in these properties to distinguish the signals in `detector.vhd` from their counterparts declared in the VHDL module `alarm_processor.vhd`. Hence the property asserts that their respective values are always equal. In the case of the modified code it is possible for them to be different, and the Solidify tool demonstrates this with the following counterexample



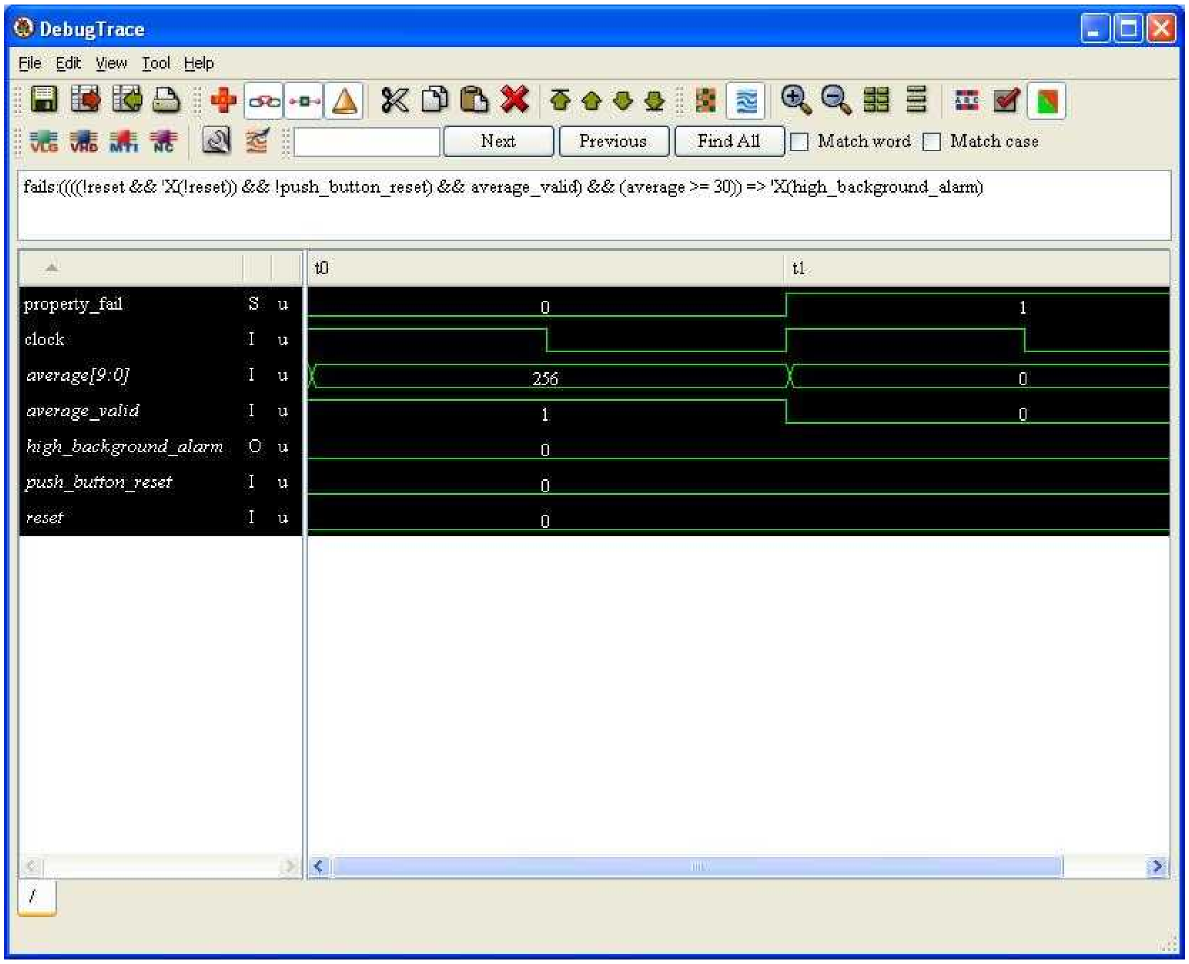
```
fails:high_background_alarm == alarms.high_background_alarm

property_fail                 S  u    1
clock                         I  u
alarms.high_background_alarm  S  u    1
high_background_alarm         O  u    0
reset                         I  u    1
```

Figure 14. Modified Design #2 Counterexample Trace

Interestingly, in the analysis of the unmodified code Solidify returns a 'vacuously true' error because the truth of the property is not dependent on the functional behaviour of the VHDL code and, as far as Solidify is concerned, this is an error. This is somewhat misleading because in this case it is neither an error nor a failure of the property.

Modified Design #3

The property which gives the relationship between the average and the `High_background_alarm` signals is derived from the value of `HIGH_ALARM_LEVEL`, i.e. 30. Similarly, the relationship between average and `Low_background_alarm` is derived from the value of `LOW_ALARM_LEVEL`. The modification to the VHDL design which uses different constant names (and values) is exposed by the analysis of `alarm_processor.vhd`.



```
fails:((((!reset && 'X(!reset)) && !push_button_reset) && average_valid) && (average >= 30)) => 'X(high_background_alarm)
```

```
property_fail          S u   0          1
clock                  I u
average[9:0]           I u   256        0
average_valid          I u   1          0
high_background_alarm  O u   0
push_button_reset      I u   0
reset                  I u   0
```

Figure 15. Modified Design #3 Counterexample Trace

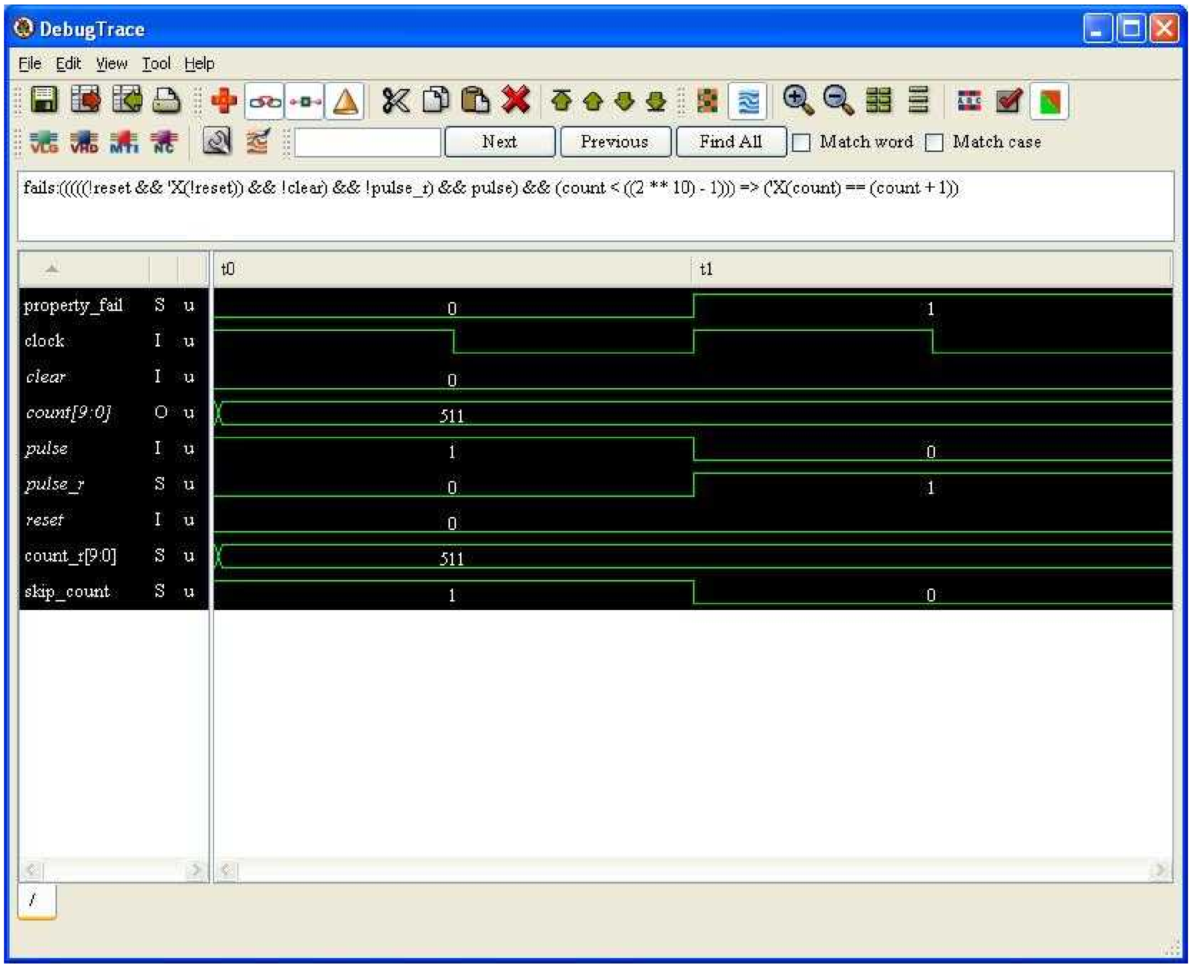
The property uses the value 30 explicitly and, hence, the counterexample shows that

… `&& average >= 30 => ‘X(High_background_alarm)`

does not always hold.

Modified Design #4

The modification which ignores every other incoming pulse from the radiation detectors by skipping increments to the `count_r` signal (and, therefore, the `Count` signal) in `pulse_counter.vhd` results in the following counterexample:



```
fails:(((((!reset && 'X(!reset)) && !clear) && !pulse_r) && pulse) && (count < ((2 ** 10) - 1))) => ('X(count) == (count + 1))
```

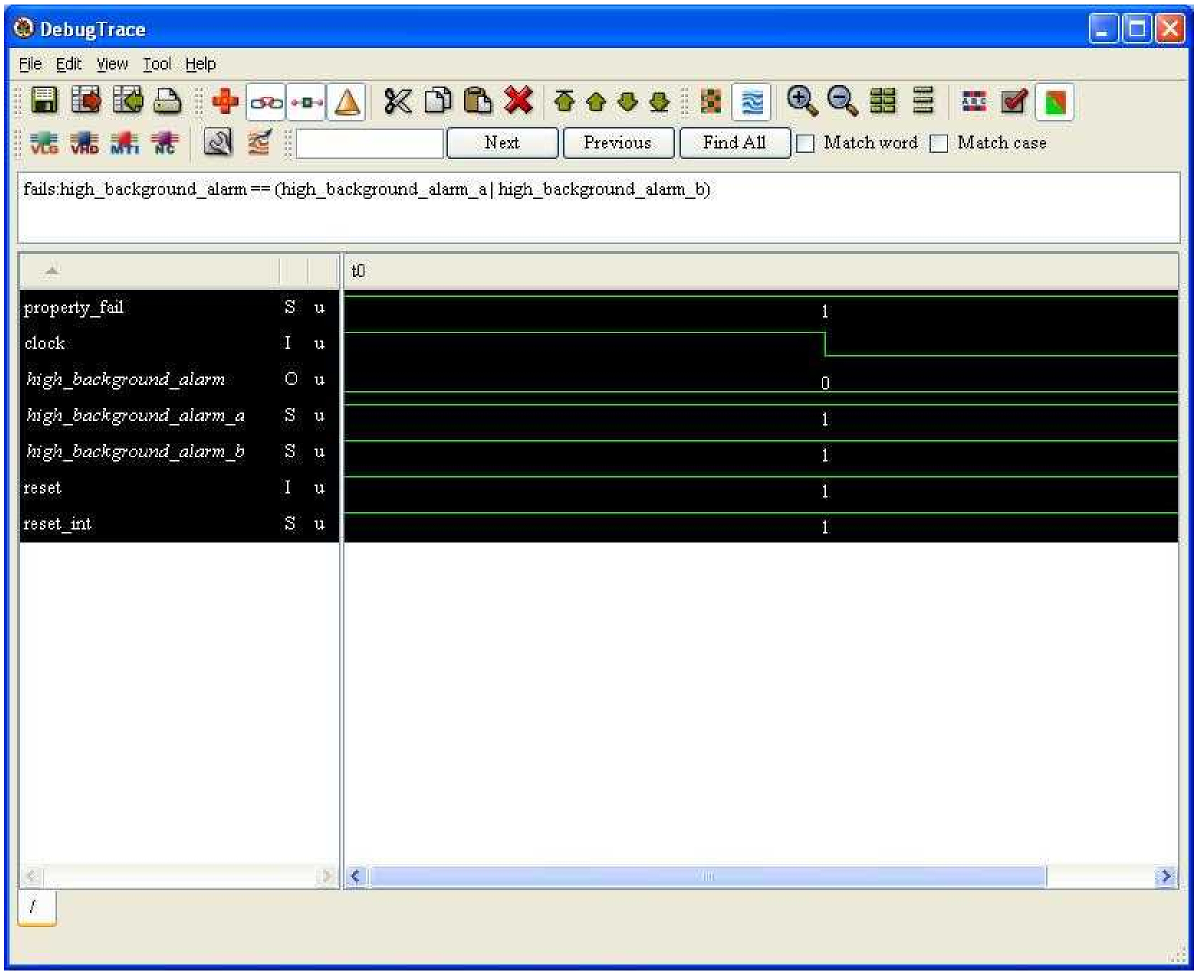
```
property_fail   S u   0 | 1
clock           I u
clear           I u   0
count[9:0]      O u   511
pulse           I u   1 | 0
pulse_r         S u   0 | 1
reset           I u   0
count_r[9:0]    S u   511
skip_count      S u   1 | 0
```

Figure 16. Modified Design #4 Counterexample Trace

The property violated by the modified VHDL code is defined to ensure that `Count` is incremented every time there is an incoming pulse. An incoming pulse is expressed in the antecedent of the property which states that `pulse_r` is low (i.e., `!pulse_r` is high) and `Pulse` is high. The counterexample demonstrates that at time step `t1` the value of count does not increment despite a low `pulse_r` and high `Pulse` at time step `t0`.

Modified Design #5

The modification which redefines the assignment of signals `High_background_alarm` and `Low_background_alarm` in terms of the exclusive-or of the corresponding alarms from the sources `A` and `B` is revealed by properties that declare the value of `High_background_alarm` and `Low_background_alarm` in terms of the disjunction of the corresponding alarms. In the case of the `High_background_alarm`, the counterexample returned by the Solidify tool is as follows:



```
fails:high_background_alarm == (high_background_alarm_a | high_background_alarm_b)
```

```
property_fail                 S  u    1
clock                         I  u
high_background_alarm         O  u    0
high_background_alarm_a       S  u    1
high_background_alarm_b       S  u    1
reset                         I  u    1
reset_int                     S  u    1
```

Figure 17. Modified Design #5 Counterexample Trace

It is obvious that this property would fail as a result of this modification.

Issues

The VHDL design considered in this work is relatively simple. The Solidify model checker analysed this design using properties that were defined over one or two time steps, as per Averant's recommended approach to 'think inductively'. As a consequence, the formal analysis using Solidify has demonstrated that it is effective in confirming and refuting properties that are necessary for authentication.

Looking ahead to the analysis of more complicated VHDL models, among the other important features of the Solidify property language HPL is the `#define` construct. This C-like macro enables user-defined properties to be named so that long HPL expressions can be simplified. More importantly, however, this construct allows properties to be defined recursively. An example of this is the definition of a macro which specifies an important part of the multiplication of polynomials in a model which performs finite field arithmetic:

```
#define product(a, f, i, m) { i == 0 ? a[m-1] & f[i] :
   {{ a[i-1] ^ (a[m-1] & f[i], product(a, f, i-1, m)}} };
```

The macro called `product` takes four parameters and specifies the multiplication of a polynomial `a[x]` (represented as an array) by the indeterminate `x` and then reducing the result modulo another polynomial `f[x]` (also represented as an array). This is a useful feature because many of the operations in finite field arithmetic are described in the literature in the form of pseudocode loops. The most natural way to represent this in HPL is by recursion.

Despite restricting the properties to simple inductive (one or two time step) formulae, model checking using Solidify on certain VHDL constructs can cause it to run out of memory. The for-loop construct is (understandably) problematic. As an example (once again from finite field arithmetic), the following code performs a polynomial reduction of a bit vector using a reduction matrix `R` which is predefined from the irreducible polynomial used for a chosen finite field:

```
gen_xors: for j in 0 to M-1 generate
  l1: process(d)
     variable aux: std_logic;
     begin
        aux := d(j);
        for i in 0 to M-2 loop
           aux := aux xor (d(M+i) and R(j)(i));
        end loop;
        c(j) <= aux;
  end process;
end generate;
```

(In this case the degree of the polynomials, `M`, is 232 but more generally it is likely that the value would be of a comparable size.) Analysis of this loop cannot be handled inductively because it does not execute over successive time steps. Hence any Solidify model checking will have to analyse the for-loop in its entirety, which is prohibitively expensive (in terms of time and memory use).

In an attempt to analyse this inductively, a macro called `loop_body` to mimic a single iteration of the loop is defined as follows

```
#define loop_body(i, j, aux, d_m_plus_i, r)
               { aux ^ (d_m_plus_i & r[j][i]) };
```

The intention of doing this is to try to enable the model checker to analyse a property of the form

```
loop_body(i, … ) => loop_body(i+1, … );
```

This `loop_body` macro is a representation of the iterations of the VHDL `for` loop in a recursive form and, hence, the associated property attempts to define an invariant for the individual iterations of the loop; in other words it is an attempt to define a loop invariant. (Since this property does not refer to the code itself, a successful property check in Solidify would return 'vacuously true' because none of the VHDL code is actually executed to determine its validity.) However, the tool still returns an out of memory error (presumably during preprocessing of the VHDL model) prior to checking this property. In conclusion it seems that, regardless of the macro and the property, Solidify seems to have problems with the for-loop construct in VHDL models. In Solidify's defence, VHDL experts probably would not implement polynomial reduction in this manner.

Refutability

The ability of a party to accept the results of a tool intended to formally verify that VHDL meets requirements could be challenging, given the nature of the expertise needed to write comprehensive and correct assertions, and the flexibility within that application. However, any finding of tamper or error would come with supporting evidence (that could be argued by the expert involved in the formal verification) that would make refutation difficult.

4.3.3 Cost Estimation

The cost of a Solidify licence is in the region of XXX, and annual support from Averant costs approximately YYY. The use of Solidify requires an expert to write assertions that are comprehensive and correct. The amount of time needed to write assertions should scale linearly with the complexity of the VHDL (as measured by the number of lines, for example). For the most efficient application of Solidify, the assertions should be written concurrently with the VHDL, though they can be written after the fact.

4.4 Formal Methods with OneSpin

Formal methods are mathematical analysis techniques that can rigorously prove correctness properties about digital systems, including hardware designs. Formal tools construct proofs of correctness by exhaustively exploring or reasoning about mathematical models of the system to analyze the correctness of a formal property that is specified by a user. Formal methods are particularly good for certain classes of properties that are difficult to rigorously verify using simulation. These include properties with qualifiers such as “if-and-only-if,” “never,” “eventually,” or “always.”

Currently available formal verification tools are very good for checking Boolean properties. They are also very good at simple algorithmic properties. Consequently, the existing formal verification tools can be quite complementary to verify those properties that are difficult to check by simulation.

Formal verification uses mathematical techniques and requires a precise, unambiguous model of the design being verified. The VHDL-level design is suitable for this analysis.

4.4.1 Theory of Operation

Traditional VHDL design verification uses simulation and testing to verify the correctness of a design. Simulation and testing provides sequences of inputs to a design and checks whether the design generates the expected outputs. The design itself is treated as a black box. Typically, various input sequences are created manually to cover the functionality of the design. A verification engineer may also tune the input sequences to test the design at anticipated corner cases. Today, constrained random simulation is also prevalent, where randomized input sequences are automatically created by a tool, subject to constraints provided by the verification engineer.

Simulation and testing show that the design works as expected for the test sequences that have been provided. But due to the large state space of digital systems, it is impossible to try every possible input sequence. In practice, testing and simulation can only try a minuscule fraction of all the possible inputs to a system. There could therefore be some untested input for which the design produces an undesirable output (such as the Pentium floating point bug[7]).

Formal verification is a white-box mathematical analysis of the internals of the design. Rather than testing the inputs of the system with various sequences, this approach creates Boolean equations of the

[7] http://download.intel.com/support/processors/pentium/sb/FDIV_Floating_Point_Flaw_Pentium_Processor.pdf

internals of the design. The equations are then "solved" to show that the expected properties hold for a set of inputs. For this report involving verification of a VHDL design, we used commercial formal verification tools. The primary technique used by these tools is model-checking. Under the hood, the solvers use Boolean satisfiability (SAT) solving techniques and binary decision diagrams (BDD).

These commercial off-the-shelf (COTS) formal verification tools require a user to input design information in VHDL or Verilog. The requirements have to then be encoded in a language such as SystemVerilog assertions (SVA). This work consists of the following:

1. Encode the requirements as SVA properties.
2. Encode any assumptions of how the inputs to the system should behave.
3. Use formal verification tools to verify the encoded properties in the implemented design, with the inputs being the design, the properties, and the assumptions. This involves some automation and some manual effort.

For the type of formal verification described here, the output of the verification tool is as follows. For each of the properties examined by the tool, the outcome can be:

1. A verification success indicating that the tool has proven that the specific property holds in the design. Sometimes, the tool can only produce a bounded-proof, i.e., the tool can guarantee that the property holds for a certain number of clock-cycles from reset, but not forever.
2. A verification failure indicating that the tool has found a problem in the property. In this case, a counter-example is produced. The counter-example is a demonstration of the failure. In this case, either the design has a problem, or the assumptions provided by the user are incorrect, or the property was not encoded properly.
3. The tool is inconclusive. In this case, either the design or the property is too complicated to arrive at a decision within a reasonable amount of time. In this case, it is often up to the verification engineer’s expertise to recast the property in a form that is easier for the tool to digest, or to break down the verification into smaller steps and guide the tool to an end.

Formal verification tools today are quite sophisticated, but they still have challenges with certain categories of verification problems such as:

- Properties that hold over a long time sequence.
- Properties that deal with memory.
- Arithmetic beyond basic integer operations.

As formal verification is performed at the VHDL level, there are some inherent limitations (these also apply to simulations):

- Certain physical implementation details cannot be verified. For example, the VHDL-level design may specify what information is stored in a register and when, but it does not specify the physical implementation of that register. The automated tools can have some latitude in implementation, as long as the implementation conforms to the VHDL-level design specification.
- Any nefarious changes made at a stage after the VHDL-level design entry cannot be detected using this technique.

A comprehensive list of the initial specifications and resulting properties we used in this authentication method can be found in Appendix D in Sections D.1 and D.2.

4.4.2 Performance Results

We applied formal verification to the representative VHDL-level design. The type of formal verification performed was to encode the "expected" properties of the design into formal properties using SVA. We used a commercial verification tool – OneSpin – to verify that the design satisfied the formal properties.

We performed an initial verification that found a few violations of some of the properties. The violations were either due to minor misinterpretations of the specifications, scenarios that would not arise in the full system, or due to steps of an algorithm not matching the expectations despite the final output being correct. There was one property violation that resulted in a change to the design.

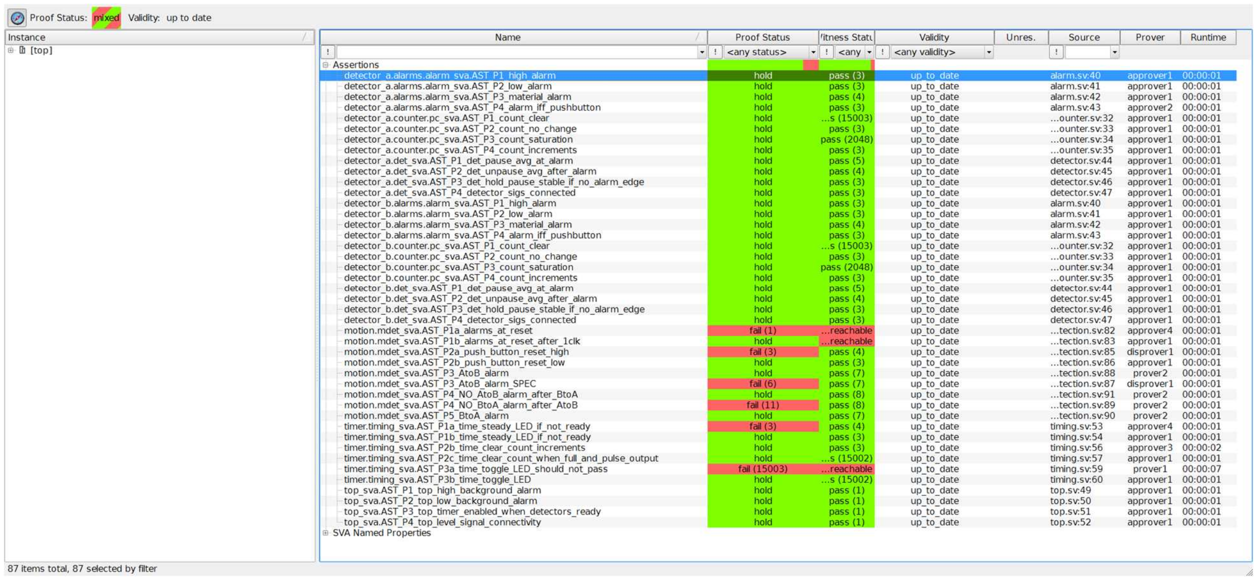
We then used the formal properties created for the original design to check the validity of different altered versions of the design. The same tool and process used for the verification of the pristine design were used on the altered designs. Examination of each alteration uncovered at least one property violation. For each property violation, a counter-example was created demonstrating how the altered design did not meet the specification.

Evaluation of Original Design

For all evaluations, we verified each module independently using specifications written by the logic designer when available. For modules without complete specifications, verification was performed against a reasonable expectation of function. For this original design verification run, there were a few counter-examples due to the following reasons:

- *Overly literal interpretation of some of the specifications.* This mismatch is easily corrected. For example, with sequential digital designs, output changes are usually updated on the next clock cycle. In some cases, when the specifications did not explicitly say that the changes would take effect on the next clock cycle, the verification assumed that it would be on the same clock cycle. This mismatch was easily fixed in verification.
- *Misinterpretation of polarity of a signal.* This mismatch again was easily fixed.
- *Incorrect expectation of the design.* After consultation with the logic designer, this inconsistency was considered to be expected behavior.
- *A scenario that, according to the logic designer, would not happen in the full design, due to the interaction of the modules.* However, the module-by-module verification did not restrict the behavior and brought up an unrealistic counter-example.
- *A property that was truly violated by the design.* The logic designer changed the design in response to this verification failure. In this case, the use of formal methods verification served as design validation.
- *Verification of individual steps of the square root algorithm.* In the first attempt, verification for the square root module was performed step-by-step following the algorithm. These counter-examples violated the algorithm step properties for input values greater than 961 due to an overflow in the delta_r integer variable. But the computation of delta_r/2-1 when delta_r = 0 restored a value of 31 to the "root" variable (root is a 6-bit variable, 31 == 2b'11111), which was the correct output for these inputs. Following the first attempt and upon consultation with the logic designer, we used a different approach that solely looked at the inputs and outputs of the square root module. In this case, the details of the computation were ignored. With the amended verification, the square root module was shown to always compute the correct value.

Some of the results can be seen in Figure 18. Note, that this list does not contain the verification of the sqrt and the count_averaging module. These had to be performed separately for efficiency.



Proof Status: mixed Validity: up to date

Instance: [top]

Assertions

Name	Proof Status	Witness Status	Validity	Unres.	Source	Prover	Runtime
detector_a.alarms.alarm_sva.AST_P1_high_alarm	hold	pass (3)	up_to_date		alarm.sv:40	approver1	00:00:01
detector_a.alarms.alarm_sva.AST_P2_low_alarm	hold	pass (3)	up_to_date		alarm.sv:41	approver1	00:00:01
detector_a.alarms.alarm_sva.AST_P3_material_alarm	hold	pass (4)	up_to_date		alarm.sv:42	approver1	00:00:01
detector_a.alarms.alarm_sva.AST_P4_alarm_iff_pushbutton	hold	pass (3)	up_to_date		alarm.sv:43	approver2	00:00:01
detector_a.counter.pc_sva.AST_P1_count_clear	hold	...s (15003)	up_to_date		...ounter.sv:32	approver1	00:00:01
detector_a.counter.pc_sva.AST_P2_count_no_change	hold	pass (3)	up_to_date		...ounter.sv:33	approver1	00:00:01
detector_a.counter.pc_sva.AST_P3_count_saturation	hold	pass (2048)	up_to_date		...ounter.sv:34	approver1	00:00:01
detector_a.counter.pc_sva.AST_P4_count_increments	hold	pass (3)	up_to_date		...ounter.sv:35	approver1	00:00:01
detector_a.det_sva.AST_P1_det_pause_avg_at_alarm	hold	pass (5)	up_to_date		detector.sv:44	approver1	00:00:01
detector_a.det_sva.AST_P2_det_unpause_avg_after_alarm	hold	pass (4)	up_to_date		detector.sv:45	approver1	00:00:01
detector_a.det_sva.AST_P3_det_hold_pause_stable_if_no_alarm_edge	hold	pass (3)	up_to_date		detector.sv:46	approver1	00:00:01
detector_a.det_sva.AST_P4_detector_sigs_connected	hold	pass (3)	up_to_date		detector.sv:47	approver1	00:00:01
detector_b.alarms.alarm_sva.AST_P1_high_alarm	hold	pass (3)	up_to_date		alarm.sv:40	approver1	00:00:01
detector_b.alarms.alarm_sva.AST_P2_low_alarm	hold	pass (3)	up_to_date		alarm.sv:41	approver1	00:00:01
detector_b.alarms.alarm_sva.AST_P3_material_alarm	hold	pass (4)	up_to_date		alarm.sv:42	approver1	00:00:01
detector_b.alarms.alarm_sva.AST_P4_alarm_iff_pushbutton	hold	pass (3)	up_to_date		alarm.sv:43	approver1	00:00:01
detector_b.counter.pc_sva.AST_P1_count_clear	hold	...s (15003)	up_to_date		...ounter.sv:32	approver1	00:00:01
detector_b.counter.pc_sva.AST_P2_count_no_change	hold	pass (3)	up_to_date		...ounter.sv:33	approver1	00:00:01
detector_b.counter.pc_sva.AST_P3_count_saturation	hold	pass (2048)	up_to_date		...ounter.sv:34	approver1	00:00:01
detector_b.counter.pc_sva.AST_P4_count_increments	hold	pass (3)	up_to_date		...ounter.sv:35	approver1	00:00:01
detector_b.det_sva.AST_P1_det_pause_avg_at_alarm	hold	pass (5)	up_to_date		detector.sv:44	approver1	00:00:01
detector_b.det_sva.AST_P2_det_unpause_avg_after_alarm	hold	pass (4)	up_to_date		detector.sv:45	approver1	00:00:01
detector_b.det_sva.AST_P3_det_hold_pause_stable_if_no_alarm_edge	hold	pass (3)	up_to_date		detector.sv:46	approver1	00:00:01
detector_b.det_sva.AST_P4_detector_sigs_connected	hold	pass (3)	up_to_date		detector.sv:47	approver1	00:00:01
motion.mdet_sva.AST_P1a_alarms_at_reset	fail (1)	...reachable	up_to_date		...tection.sv:82	approver4	00:00:01
motion.mdet_sva.AST_P1b_alarms_at_reset_after_1clk	hold	...reachable	up_to_date		...tection.sv:83	approver1	00:00:01
motion.mdet_sva.AST_P2a_push_button_reset_high	fail (3)	pass (4)	up_to_date		...tection.sv:85	disprover1	00:00:01
motion.mdet_sva.AST_P2b_push_button_reset_low	hold	pass (3)	up_to_date		...tection.sv:86	approver1	00:00:01
motion.mdet_sva.AST_P3_AtoB_alarm	hold	pass (7)	up_to_date		...tection.sv:88	prover2	00:00:01
motion.mdet_sva.AST_P3_AtoB_alarm_SPEC	fail (6)	pass (7)	up_to_date		...tection.sv:87	disprover1	00:00:01
motion.mdet_sva.AST_P4_NO_AtoB_alarm_after_BtoA	hold	pass (8)	up_to_date		...tection.sv:91	prover2	00:00:01
motion.mdet_sva.AST_P4_NO_BtoA_alarm_after_AtoB	fail (11)	pass (8)	up_to_date		...tection.sv:89	prover2	00:00:01
motion.mdet_sva.AST_P5_BtoA_alarm	hold	pass (7)	up_to_date		...tection.sv:90	prover2	00:00:01
timer.timing_sva.AST_P1a_time_steady_LED_if_not_ready	fail (3)	pass (4)	up_to_date		timing.sv:53	approver4	00:00:01
timer.timing_sva.AST_P1b_time_steady_LED_if_not_ready	hold	pass (3)	up_to_date		timing.sv:54	approver1	00:00:01
timer.timing_sva.AST_P2b_time_clear_count_increments	hold	pass (3)	up_to_date		timing.sv:56	approver3	00:00:02
timer.timing_sva.AST_P2c_time_clear_count_when_full_and_pulse_output	hold	...s (15002)	up_to_date		timing.sv:57	approver1	00:00:01
timer.timing_sva.AST_P3a_time_toggle_LED_should_not_pass	fail (15003)	...reachable	up_to_date		timing.sv:59	prover1	00:00:07
timer.timing_sva.AST_P3b_time_toggle_LED	hold	...s (15002)	up_to_date		timing.sv:60	approver1	00:00:01
top_sva.AST_P1_top_high_background_alarm	hold	pass (1)	up_to_date		top.sv:49	approver1	00:00:01
top_sva.AST_P2_top_low_background_alarm	hold	pass (1)	up_to_date		top.sv:50	approver1	00:00:01
top_sva.AST_P3_top_timer_enabled_when_detectors_ready	hold	pass (1)	up_to_date		top.sv:51	approver1	00:00:01
top_sva.AST_P4_top_level_signal_connectivity	hold	pass (1)	up_to_date		top.sv:52	approver1	00:00:01

SVA Named Properties

87 items total, 87 selected by filter

Figure 18. Verification Results of the Unmodified Reference Design

Eventually, all modules passed verification without failing the proof due to complexity or generating a counter-example.

Evaluation of Modified Designs

Modified Design #1

This design has an alternate version of the sqrt module. Running the verification in D.2.7 shows a counter-example for the property in D.2.7.2. The counter-example (in Figure 19) shows the most trivial case where for an input of "0", the output is "F".

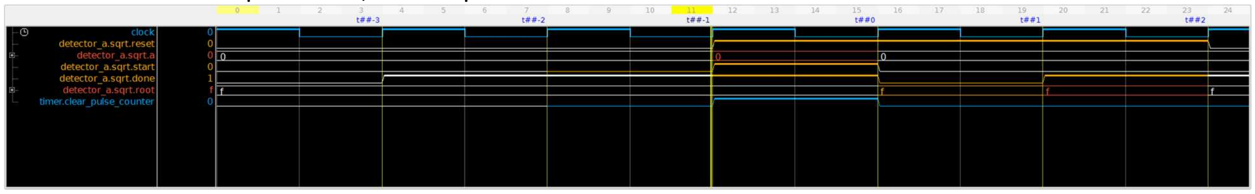


Figure 19. Counter-example for Modified Design #1

Modified Design #2

This design has an alternate version of the detector module. Running the verification in D.2.3 shows a counter-example for the connectivity property in D.2.3.4. The counter-example shows that the background_alarm signals (both high and low) are different from the background alarm signals coming from the alarm_processor sub-module within the detector. They are expected to be the same. This discrepancy is highlighted in red in Figure 20.

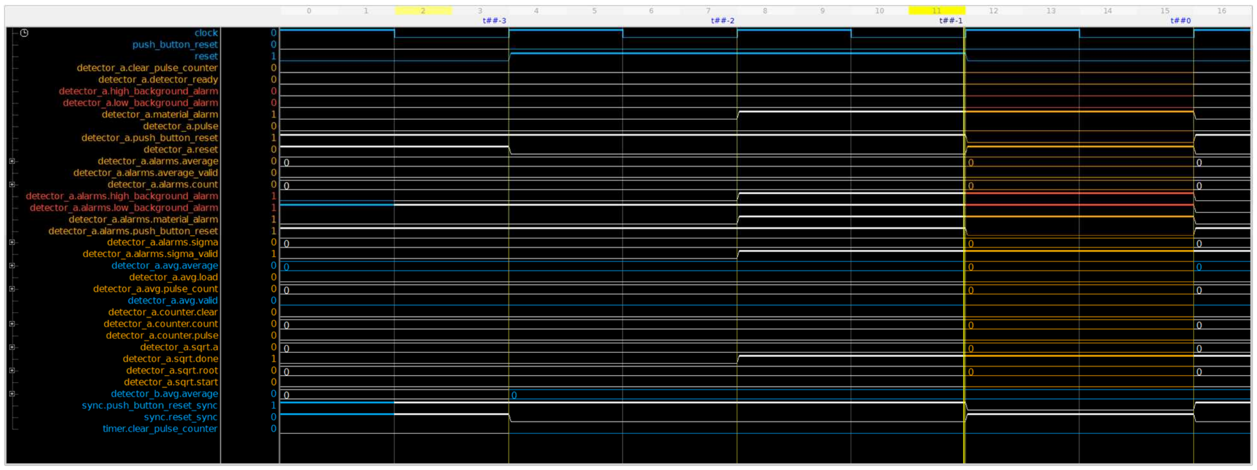

clock
push_button_reset
reset
detector_a.clear_pulse_counter
detector_a.detector_ready
detector_a.high_background_alarm
detector_a.low_background_alarm
detector_a.material_alarm
detector_a.pulse
detector_a.push_button_reset
detector_a.reset
detector_a.alarms.average
detector_a.alarms.average_valid
detector_a.alarms.count
detector_a.alarms.high_background_alarm
detector_a.alarms.low_background_alarm
detector_a.alarms.material_alarm
detector_a.alarms.push_button_reset
detector_a.alarms.sigma
detector_a.alarms.sigma_valid
detector_a.avg.average
detector_a.avg.load
detector_a.avg.pulse_count
detector_a.avg.valid
detector_a.counter.clear
detector_a.counter.count
detector_a.counter.pulse
detector_a.sqrt.a
detector_a.sqrt.done
detector_a.sqrt.root
detector_a.sqrt.start
detector_b.avg.average
sync.push_button_reset_sync
sync.reset_sync
timer.clear_pulse_counter

Figure 20. Counter-example for Modified Design #2

Modified Design #3

This design has an alternate version of the alarm_processor module. Running the verification in D.2.5 shows counter-examples for two properties (D.2.5.1 and D.2.5.2). For the high alarm, Figure 21 shows that the design has reached the conditions for triggering the alarm, but the alarm remains negated. Figure 22 shows similar conditions failing to activate the low alarm. For each figure, see the red traces that outline where the alarm signals deviate from the expected property.

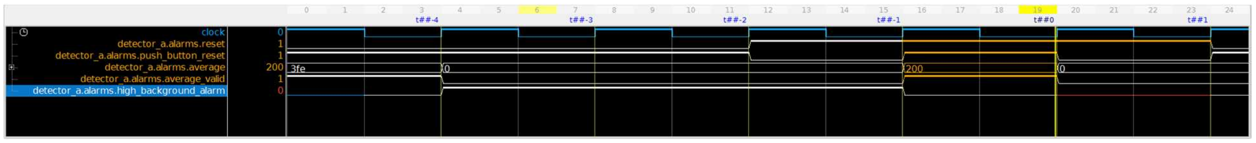

clock
detector_a.alarms.reset
detector_a.alarms.push_button_reset
detector_a.alarms.average
detector_a.alarms.average_valid
detector_a.alarms.high_background_alarm
3fe
200

Figure 21. Counter-example for Modified Design #3 (High Alarm)

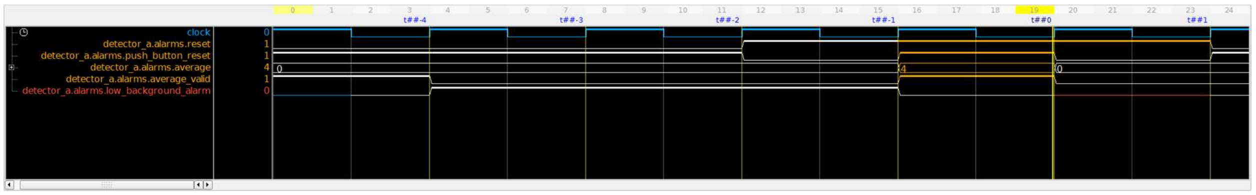

clock
detector_a.alarms.reset
detector_a.alarms.push_button_reset
detector_a.alarms.average
detector_a.alarms.average_valid
detector_a.alarms.low_background_alarm

Figure 22. Counter-example for Modified Design #3 (Low Alarm)

Modified Design #4

This design has an alternate version of count_averaging module. Running the verification in D.2.4 shows a counter-example for the property in D.2.4.4. This property describes the proper conditions needed for the pulse counter to increment. Figure 23 presents the counter-example. It shows that under certain conditions, the counter is not incremented. The counter-example also automatically picks out a signal called skip_count that is related to this failure.

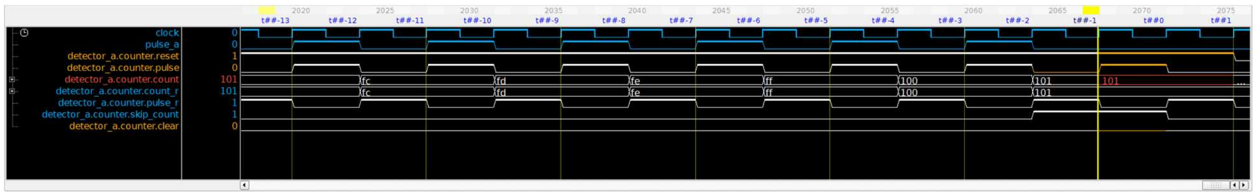


Figure 23. Counter-example for Modified Design #4

Modified Design #5

This design has an alternate version of the top_level module. Running the verification in D.2.9 shows counter-examples for two properties (D.2.9.1 and D.2.9.2). These properties ensure that the top level alarm is set when either of the detectors alarm. The counter-examples (Figure 24 and Figure 25) show that these top level signals are not an ORed version of the detector alarms. See the red outline of the expected value of the signal.

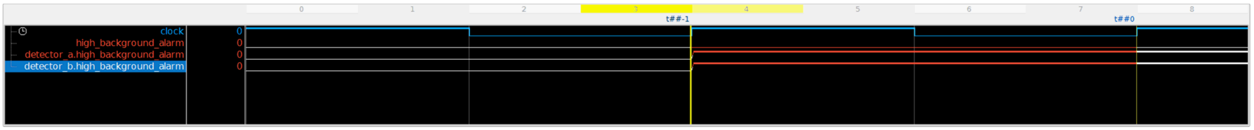

clock
high_background_alarm
detector_a.high_background_alarm
detector_b.high_background_alarm
t##-1
t##0

Figure 24. Counter-example for Modified Design #5 (High Alarm)

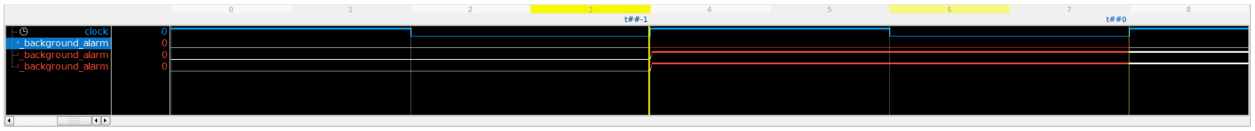


Figure 25. Counter-example for Modified Design #5 (Low Alarm)

Refutability

The goal of this work was to verify the design using formal properties and to see if the created formal properties could detect alterations in the design. Following are some observations from this work pertaining to these goals:

- **Completeness of specifications:** Creating precise formal properties requires accompanying precision in the language of the specifications. Areas of the specification that are ambiguous or admit a range of behavior can provide room for malicious alteration. Ambiguous specifications can allow alterations of the design that seemingly conform to the specification but provide undesired behavior.
- **Complexity of the design:** Formal verification tools today are limited in the complexity of the design and the complexity of the property that can be examined. Malicious alteration of the design to add sufficient gratuitous complexity would make it unanalyzable and the verification results inconclusive.
- **Independence of the verification:** In practice, specification documents do not contain every bit of detail required to specify (or verify) a digital design. In these cases, the verification engineer turns to the design itself to provide this information. This approach risks losing the independence of the verification and biasing the verification properties.
- **Module-level verification vs system-level properties:** Formal verification tools are good for verifying individual functional modules within the design. The assembly of these modules into a cohesive and correct design depends on the designer. There are other formal techniques to help with this higher-level verification, but the standard formal tools meant for HDL are limited in this capability.
- **Completeness of verification properties:** With formal property verification, if the specification is incomplete (there are unspecified portions of the design), the properties are incomplete. Verification of an incomplete set of properties does not prevent the malicious presence of extra functionality in the unchecked sections while still satisfying the existing formal properties.

4.4.3 Cost Estimation

The software tools used for this verification was DV-Verify by OneSpin. Formal property verification was performed using this tool and the properties listed in earlier sections. The cost of this tool depends on the licensing model. A short-term license costs tens of thousands of dollars. The effort taken to setup this design, write the properties against the original requirements, debug, and generate the documentation was approximately 80 person hours of a formal verification expert.

4.5 Formal Equivalence Check Survey

Equivalence checking in general determines whether two Boolean functions are equivalent. In the context of digital hardware this means deciding whether two combinational circuits are equivalent. Typically, of course, circuits are not solely combinational because they also contain state registers and feedback loops. In order to perform equivalence checking on these circuits it would be necessary to remove such artefacts. Equivalence means that, for all possible inputs, the outputs of the two functions (or circuits) are equal. Testing every possible input/output pair is infeasible for circuits in general but by exploiting shared structure within the circuits the problem becomes less challenging.

An example combined approach (originating from the methods described in Kuelmann, et al.[8]) illustrates several of the common techniques used in checking equivalence. The approach begins by constructing an and-Inverter graph (AIG) from the circuits. An AIG is a directed acyclic graph consisting of vertices (nodes) and directed edges. Each node is either an input node (no incoming edges), a constant node (representing a logical 0 or 1), or an ‘and’ node with two incoming edges. The outgoing edges from the ‘and’ node convey the result of the logical operation on the values of its incoming edges. Edges can be inverted to represent the logical negation of Boolean values passing along the edges. This enables the representation of other logical primitives (in addition to the functionality provided by the ‘and’ node).

Structural hashing during the construction of the AIG can be used to simplify the resulting structure. A check is made prior to the introduction of an ‘and’ node (with its two incoming edges) in case a node with the same incoming edges already exists, and can be reused. Structural hashing therefore eliminates one specific kind of redundancy in the AIG. Other types of structural hashing have been devised to eliminate other kinds of redundancy. It is possible to determine equivalence from structural hashing if the resulting AIG constructs a single ‘and’ node with two outgoing edges that produce the outputs of the two original circuits. (Alternatively, the output edges can be connected by a representation of an exclusive nor (XNOR) gate to determine whether the circuit can be simplified to a constant ‘1’ output.)

More often than not, structural hashing will be insufficient as a means of equivalency checking. The structural hashing in Kuelmann, et al., is supplemented by *Binary Decision Diagram* (BDD) *sweeping*. A BDD is canonical representation of a Boolean function. Each node within an AIG has an associated BDD which describes the function of the AIG from the AIG’s input nodes. The ‘sweep’ refers to the systematic construction of BDDs from the input nodes to the outputs. If, during the sweep, nodes with identical BDDs are identified then they can be merged, and the resulting AIG can then be subjected to structural hashing in an attempt to simplify the graph further. This approach can be enhanced by ‘cutting’ the AIG at strategic points so that BDD analysis can take place from intermediate ‘cut frontiers’ as well as its input nodes. This enables deeper exploration of large AIGs.

[8] A. Kuelmann, V. Paruthi, F. Krohm, M. Ganai: Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 21, No. 12, 2002.

As a further enhancement, the approach taken by Kuelmann, et al., incorporates a method of *satisfiability* (SAT) *solving* AIGs. In a manner similar to traditional SAT solving, this approach attempts to find a consistent assignment (with respect to the Boolean function represented by the AIG) of Boolean values to the nodes that result in the assignment of a 'target' node with a value 1, thereby demonstrating that the AIG is satisfiable. This works by starting at the target node and propagating backwards to determine all possible assignments. (This also requires a certain amount of forward propagation to ensure consistency with other parts of the AIG.) Any conflicting assignments cause the algorithm to 'backtrack' in an attempt to find alternative non-conflicting assignments. Presumably, in an equivalence checking context, SAT solving is used to show that a consistent assignment in which the output is '0' is unsatisfiable. Hence the output is always '1', thereby demonstrating equivalence.

By limiting the amount of backtracking allowed in the SAT solving procedure, and limiting the depth of the BDD sweep, it is possible to have a combined approach in which BDD sweeping (including structural re-hashing) and SAT solving are interleaved until the AIG can be simplified to the extent that SAT solving succeeds.

Random simulation is yet another method that can assist in equivalency checking. By inputting random vectors to the AIG, 'candidate equivalent pairs' of nodes are identified. The techniques described above can then focus on the candidate equivalent pairs within the AIG to determine actual equivalence and simplify the graph.

In addition to OneSpin and Formality (described in the sections below), the methods described above have been implemented in IBM's Verity tool[9]. Other tools that list equivalence checking explicitly among their capabilities are ABC and Cryptol. Berkeley's ABC tool[10] provides synthesis and verification of binary sequential circuits in synchronous hardware designs. Verification is based on AIG representations of circuits. It does not seem to accept VHDL designs as input, but it does accept (among other things) a subset of electronic design interchange format (EDIF) and structural Verilog. Cryptol[11] is a functional language tailored for cryptographic algorithms but is accompanied by a suite of tools for verification. The reason it is mentioned in the context of this work is because it targets FPGAs. Equivalence checking is achieved by translating Cryptol into a symbolic bit-vector (SBV) program which maps input bit vectors to output bit vectors. The SBV program can be analysed for equivalence by using a SAT solver or, more recently, by using a satisfiability modulo theories (SMT) solver. SMT tools extend SAT solving by allowing more exotic mathematical expressions (rather than just Boolean expressions) to be checked for satisfiability. One important advantage of SMT solvers over SAT solvers for equivalence checking is that the former can reason about bit vectors as well as simple Boolean values. This can be used for higher-level structural reasoning prior to bit-level analysis.

SMT solvers such as Yices[12] list equivalence checking among their capabilities but this seems to be only inasmuch as SMT solving in general is capable of equivalency checking. In other words, they do not seem to be tailored specifically to equivalence checking.

[9] A. Kuelmann, A. Srinivasan, D. LaPotin: Verity – a Formal Verification Program for Custom CMOS Circuits. IBM Journal on Research and Development, 1995.
[10] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www/eecs/berkeley.edu/~alanmi/abc
[11] Galois, Inc. Cryptol: The Language of Cryptography. http://cryptol.net
[12] SRI International. The Yices SMT Solver. http://yices.csl.sri.com

4.6 Formal Equivalence Check with OneSpin

Equivalence checking (EC) is a method by which two designs, usually the second being derived from the first, are compared for formal equivalence. Two sequential designs are said to be formally equivalent if they produce identical outputs on each clock cycle given an identical set of inputs.

We used the 360 EC product from OneSpin Solutions Gmbh[13] to perform EC between the source RTL (in VHDL format) and the placed-and-routed netlist (in Verilog format). 360 EC offers both combinational and sequential equivalence checking capabilities (see section 4.6.1) and is designed to work with most FPGA logic families and synthesis engines. We used both Xilinx ISE and Synopsys Synplify Pro to synthesize the RTL and Xilinx tools to place and route the design.

We tested several variants of the synthesized design, as shown in Table 4. The first four variants represent the original design, with different synthesis tools and optimization options. To be considered successful, EC must correctly report the netlist equivalent to the RTL, with no caveats. The remaining variants represent five different modifications to the design, which EC is required to detect. To be considered successful, EC must not only report the designs as non-equivalent but also provide sufficient specificity to allow the analyst to correctly identify the modified function apart from the remaining circuits. A change is "fully isolated" if the analyst correctly deduces the exact function of the change. A change is "partially isolated" if the analyst correctly identifies the functional unit(s) affected by the change.

Table 4. Equivalence Checking (EC) Results

Design Variant	Outcome	Runtime	Result
Basic Optimizations (ISE)	Combinational equivalence passed	2 min 34 sec	Success
Basic + Register Duplication (ISE)	Combinational equivalence failed; hybrid sequential equivalence passed	3 min 10 sec	Success
Basic Optimizations (Synplify Pro)	Combinational equivalence passed	2 min 0 sec	Success
Basic + Retiming (Synplify Pro)	Combinational equivalence failed; sequential equivalence inconclusive	1 hour	Failure
Modified Design #1	Change detected and partially isolated	N/A	Success
Modified Design #2	Change detected and fully isolated	N/A	Success
Modified Design #3	Change detected and partially isolated	N/A	Success
Modified Design #4	Change detected and partially isolated	N/A	Success
Modified Design #5	Change detected and fully isolated	N/A	Success

4.6.1 Theory of Operation

Given a pair of similar sequential designs having corresponding inputs and outputs, the designs are considered *equivalent* if they produce the same logic value on each corresponding output given identical corresponding inputs and state history. This equivalence checking can be seen in the miter configuration of Figure 26, in which n signals are each connected to the corresponding input of a pair of designs-under-test (DUT). Here it is assumed each DUT is a sequential circuit with inputs aligned to a common clock of sufficient period. That is, each DUT x consists of an input vector $\boldsymbol{I}_x \in \{0,1\}^n$, an output vector

[13] https://www.onespin.com/

$\boldsymbol{O}_x \in \{0,1\}^m$, a state vector $\boldsymbol{S}_x$, and a clock, which advances the state of the design on every period as a function of the current state, inputs and outputs: $\boldsymbol{S}_x, \boldsymbol{O}_x \leftarrow \boldsymbol{f}_x(\boldsymbol{I}_x, \boldsymbol{S}_x, \boldsymbol{O}_x)$. To create the miter, we constrain $\boldsymbol{I}_1 = \boldsymbol{I}_2 = \boldsymbol{I} \in \{0,1\}^n$ and define $\boldsymbol{R} = \boldsymbol{O}_1 \oplus \boldsymbol{O}_2$, the exclusive or (XOR) of each pair of outputs. Each XOR gate produces 0 when the corresponding outputs match and 1 when they mismatch. Under these conditions, the DUTs are equivalent if $\boldsymbol{R} = \boldsymbol{0}$ (all zeros) following all possible sequences of $\boldsymbol{I}$ applied to common starting state.

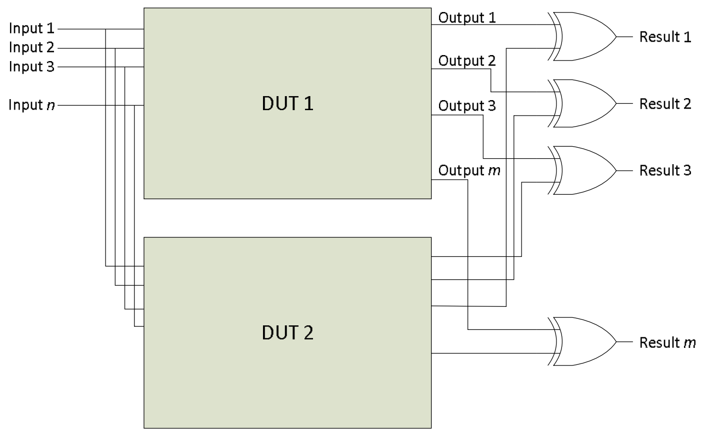


Figure 26. Miter Circuit for Equivalence Checking

Two types of EC are available for sequential designs: combinational equivalence checking (CEC) and sequential equivalence checking (SEC), which differ in how the starting state is defined.

In SEC, the starting state is a reset condition or other state of importance, from which the design can be said to behave predictably to a subsequent input stimulus. The input stimulus can span an arbitrary number of clock cycles and is assumed to be infinite (though the range of behavior of a design is certainly not infinite). The state term $\boldsymbol{S}_x$ is encapsulated in the composition of all previous inputs, and the output term at time k becomes $\boldsymbol{O}_x(k) \leftarrow \boldsymbol{f}_x(\boldsymbol{I}_x(0), \boldsymbol{I}_x(1), \ldots, \boldsymbol{I}_x(k-1))$. This black-box definition allows DUT 1 and DUT 2 to be implemented quite differently and still be equivalent, as long as they produce identical outputs given identical inputs. However, this type of equivalence is very difficult to prove, as it involves enumerating or inferring all possible responses of each design to an unbounded stimulus, which scales exponentially with design size. Modern SEC tools use a number of optimizations in attempt to reduce complexity of the problem, but in general SEC remains intractable for designs of moderate complexity.

In CEC, we apply a series of relaxations to greatly simplify the computation. First, we assume that the DUTs are composed of $l + m$ equivalent pairs of state and output bits $\boldsymbol{B}_2 \equiv \boldsymbol{B}_1 \in \{0,1\}^{l+m}$ under an equivalence function $\boldsymbol{\mathcal{M}}$, which is called a *map*. Second, we assume an arbitrary initial state $\boldsymbol{B}_2 = \boldsymbol{\mathcal{M}}(\boldsymbol{B}_1) = \boldsymbol{\mathcal{M}}(\boldsymbol{B}) \in \{0,1\}^{l+m}$, and consider only the space of possible transitions from this initial state to the immediate next state. We assert that by covering all possible transitions from any arbitrary state (including ones that are not logically reachable), we cover a superset of the design state space. The goal in CEC, then, is to find $\boldsymbol{\mathcal{M}}$ such that $\boldsymbol{f}_2(\boldsymbol{I}_2, \boldsymbol{B}_2) = \boldsymbol{\mathcal{M}}(\boldsymbol{f}_1(\boldsymbol{I}_1, \boldsymbol{B}_1))$. Together these assumptions eliminate the costly enumeration of reachable states and allow CEC to scale linearly with design size. However, they tend to hold only when one DUT is derived from the other, and even then only under a limited set of allowed transformations. Some sequential optimizations supported by FPGA tools, such as register retiming and duplication, break combinational equivalence. Additionally, netlists that follow divergent implementation paths from their source RTL are not guaranteed to be combinationally equivalent due to differing optimizations around “don’t care conditions” and transitions from illegal/unreachable states. This equivalence is illustrated graphically in Figure 27.

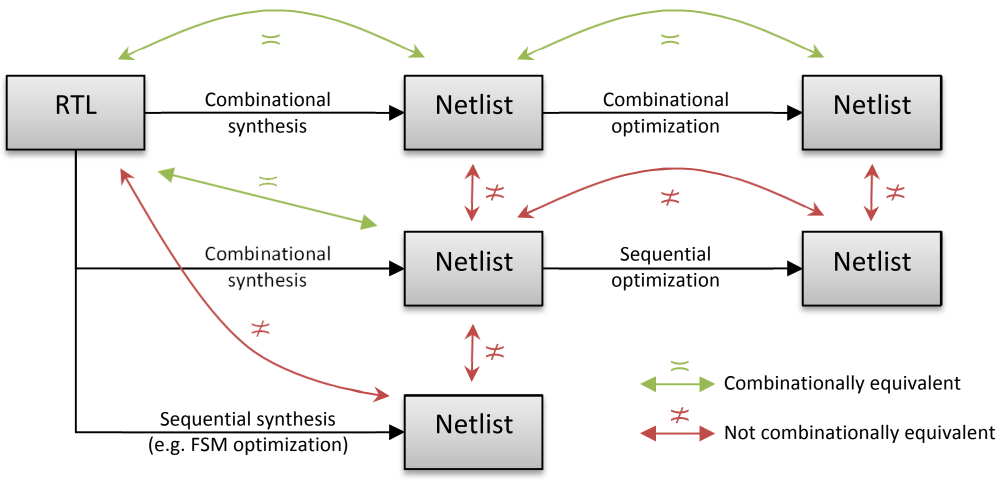

RTL
Combinational synthesis
Netlist
Combinational optimization
Netlist
Combinational synthesis
Netlist
Sequential optimization
Netlist
Sequential synthesis (e.g. FSM optimization)
Netlist
Combinationally equivalent
Not combinationally equivalent

Figure 27. Synthesis Transformations and their Combinational Equivalence

Thus CEC is conservative in that if a design is proven combinationally equivalent, it is also sequentially equivalent; conversely, designs that are sequentially equivalent are not necessarily combinationally equivalent. It is also conservative in the sense that transitions from illegal and unreachable states are compared, whereas in SEC they may be ignored or proven vacuously. This minor annoyance becomes an asset when designing for fault tolerance. The remainder of this section describes the mapping process for CEC and computation of initial state.

Internal state mappings may be simple ($S_{1,i} = S_{2,j}$), inverted ($S_{1,i} = \neg S_{2,j}$), or complex ($\boldsymbol{S_{1,i} = f(S_{2,j})}$), where $\boldsymbol{f}$ is an arbitrary one-to-one function). Output mappings must be simple; otherwise the design cannot be said to be sequentially equivalent. Additionally, a design may contain *relations*, which equate its state bits with a constant value (e.g. $S_{x,i} = 0$) or with other state bits in the same design (e.g. $S_{x,i} = S_{x,j}$). 360 EC uses proprietary proof-based methods to infer mappings and relations and also employs heuristics to prune the search space: name-based mapping and structural/functional mapping. Name-based mapping takes advantage of naming practices of the synthesis tools, which create unique, predictable instance names for each synthesized RTL register. If two netlist instances have the same or similar name, or if the netlist has an instance name similar to an RTL register, the pair will be mapped during this step. (The tool uses a configurable regular expression to determine similarity). Following name-based mapping, the tool then maps pairs of state bits having similar fanin and fanout structure or function during the structural/functional mapping step. Heuristic methods are fast and efficient, but may sometimes result in incorrect mappings, which lead to CEC failures. Proof-based mapping is the final step before comparison; as the name implies, any mappings or relations created during this step are pre-proven and will pass CEC. If any bits exist without a mapping or relation, they will be treated as free variables and able to take on either value in $\{0,1\}$ for producing a counterexample (proof of non-equivalence). Free variables thus usually result in CEC failures on every state bit they influence.

360 EC computes the initial state (e.g., the reset state) of each design to determine the starting state for optional SEC but also to determine if each heuristic mapping is simple or inverted. If a mapped pair resets to the same value, a simple mapping is inferred; if they initialize to opposite values, an inverted mapping is inferred. In some cases, the initial state cannot be determined for all state bits, such as in complex reset schemes and uninitialized memories. In these cases, some mappings may be inferred incorrectly, which will cause CEC failures. False CEC failures due to invalid mappings may be resolved by importing initial state from simulation, changing the order of initial state computation relative to

mapping, or skipping some mapping steps altogether (like structural/functional). Alternatively, SEC can be used to verify the subset of bits that fail CEC, assuming the sequential logic is relatively simple.

4.6.2 Performance Results

For the representative design used in this feasibility study, we showed that CEC is feasible provided that certain sequential optimizations are avoided. Register balancing (ISE) and register retiming (Synplify Pro) both caused false equivalence failures that could not be reconciled, limiting the technique's usefulness at detecting and isolating changes in the presence of these optimizations. In earlier versions of the tool, it inferred incorrect mappings, causing CEC failures, e.g., a pair of simple mappings where a complex mapping should have been. These appear to have been corrected in the latest release at the time of this writing (version 2016_06).

We found full-design SEC to be intractable for designs of this nature due to the size of the state space. However, 360 EC supports a hybrid equivalence check, whereby it first attempts CEC, followed by SEC in an effort to prove equivalence points that failed combinationally. This hybrid equivalence checking was found to be successful in reversing false CEC failures due to register duplication in ISE.

Evaluation of Original Design with Basic Optimizations (ISE)

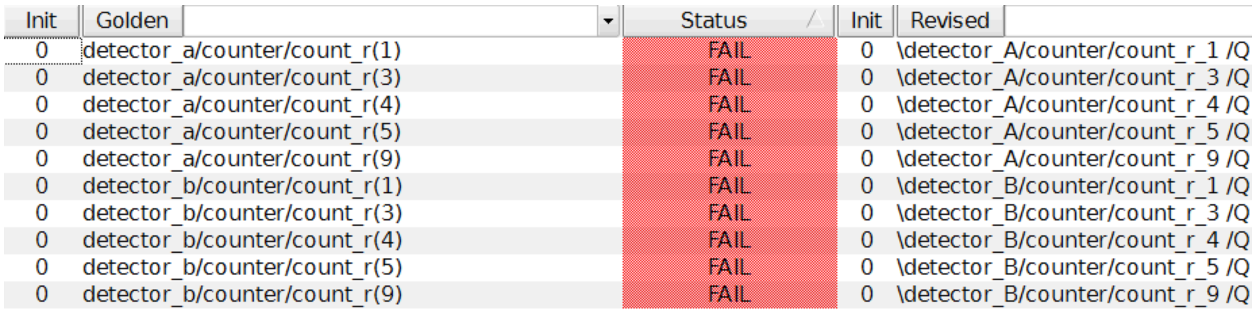
The original RTL was synthesized in Xilinx ISE using the default optimizations (without register balancing), then placed and routed in Xilinx ISE. We compared the resulting netlist to the original RTL using 360 EC in combinational equivalence mode. In 360 EC, the netlist inputs and outputs were mapped by name to the corresponding inputs and outputs of the RTL. The internal states of the design were mapped using the tool's proof-based algorithm. All 41,175 mapped state pairs and 5 output pairs were determined to be equivalent. The total runtime was 2 minutes and 34 seconds.

Command sequence:

```
map -input -output
compute_initial_state
compute_state_relations
compare
```

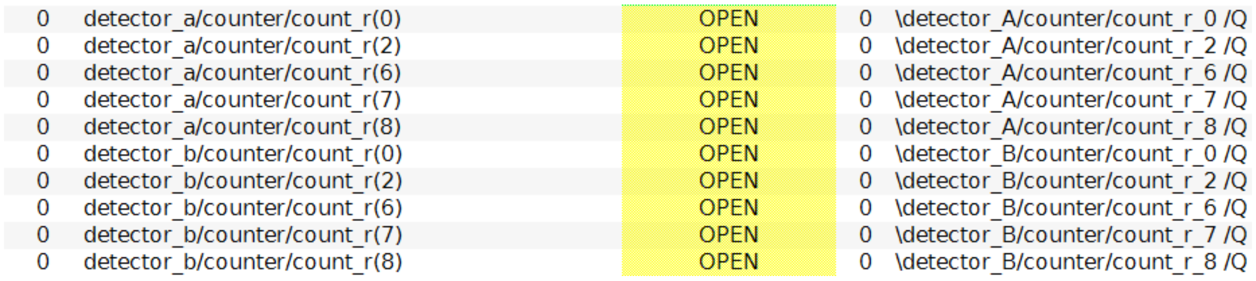
Evaluation of Original Design with Register Balancing (ISE)

We repeated the previous experiment, this time allowing ISE to perform register balancing. The tool mapped 191 internal states by name and another 40,986 using proof-based methods. However, there were 20 states in the netlist that could not be mapped to RTL registers, resulting in 10 failed combinational compare points and another 10 states that could not be compared combinationally, as shown below.



Init	Golden	Status	Init	Revised
0	detector_a/counter/count_r(1)	FAIL	0	\detector_A/counter/count_r_1 /Q
0	detector_a/counter/count_r(3)	FAIL	0	\detector_A/counter/count_r_3 /Q
0	detector_a/counter/count_r(4)	FAIL	0	\detector_A/counter/count_r_4 /Q
0	detector_a/counter/count_r(5)	FAIL	0	\detector_A/counter/count_r_5 /Q
0	detector_a/counter/count_r(9)	FAIL	0	\detector_A/counter/count_r_9 /Q
0	detector_b/counter/count_r(1)	FAIL	0	\detector_B/counter/count_r_1 /Q
0	detector_b/counter/count_r(3)	FAIL	0	\detector_B/counter/count_r_3 /Q
0	detector_b/counter/count_r(4)	FAIL	0	\detector_B/counter/count_r_4 /Q
0	detector_b/counter/count_r(5)	FAIL	0	\detector_B/counter/count_r_5 /Q
0	detector_b/counter/count_r(9)	FAIL	0	\detector_B/counter/count_r_9 /Q

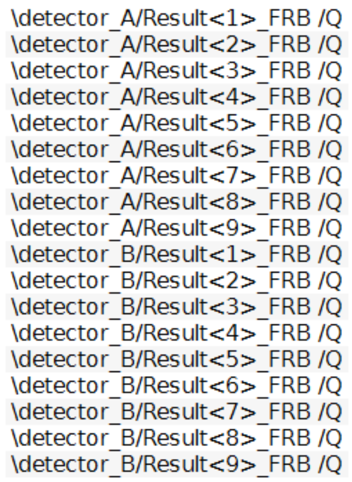
Figure 28. Failed Combinational Compare Points of Original Design with Register Rebalancing (ISE)



0	detector_a/counter/count_r(0)	OPEN	0	\detector_A/counter/count_r_0 /Q
0	detector_a/counter/count_r(2)	OPEN	0	\detector_A/counter/count_r_2 /Q
0	detector_a/counter/count_r(6)	OPEN	0	\detector_A/counter/count_r_6 /Q
0	detector_a/counter/count_r(7)	OPEN	0	\detector_A/counter/count_r_7 /Q
0	detector_a/counter/count_r(8)	OPEN	0	\detector_A/counter/count_r_8 /Q
0	detector_b/counter/count_r(0)	OPEN	0	\detector_B/counter/count_r_0 /Q
0	detector_b/counter/count_r(2)	OPEN	0	\detector_B/counter/count_r_2 /Q
0	detector_b/counter/count_r(6)	OPEN	0	\detector_B/counter/count_r_6 /Q
0	detector_b/counter/count_r(7)	OPEN	0	\detector_B/counter/count_r_7 /Q
0	detector_b/counter/count_r(8)	OPEN	0	\detector_B/counter/count_r_8 /Q

Figure 29. Combinational Comparisons in Original Design with Register Rebalancing (ISE) that Could Not Complete

The above failures were due to the following duplicated registers, which have no combinational equivalents in RTL:



```
\detector_A/Result<1>_FRB /Q
\detector_A/Result<2>_FRB /Q
\detector_A/Result<3>_FRB /Q
\detector_A/Result<4>_FRB /Q
\detector_A/Result<5>_FRB /Q
\detector_A/Result<6>_FRB /Q
\detector_A/Result<7>_FRB /Q
\detector_A/Result<8>_FRB /Q
\detector_A/Result<9>_FRB /Q
\detector_B/Result<1>_FRB /Q
\detector_B/Result<2>_FRB /Q
\detector_B/Result<3>_FRB /Q
\detector_B/Result<4>_FRB /Q
\detector_B/Result<5>_FRB /Q
\detector_B/Result<6>_FRB /Q
\detector_B/Result<7>_FRB /Q
\detector_B/Result<8>_FRB /Q
\detector_B/Result<9>_FRB /Q
```

Figure 30. Registers that Caused Failure in Original Design with Register Rebalancing (ISE)

In such cases, 360 EC offers the ability to re-test the failing compare points using sequential equivalence. This limited re-testing is more tractable than checking sequential equivalence of the entire design because it is able to leveraging the existing state mappings that have already been proven.

Using sequential equivalence, the tool successfully proved the remaining 20 compare points. Thus, using a combination of combinational and sequential equivalence, the netlist was proven equivalent to the RTL. Total runtime was 3 minutes and 10 seconds.

Command sequence:

```
map -input -output
compute_initial_state
map -nameonly
compute_state_relations
compare
reset_compare_status -fail
compare -sequential
```

Evaluation of Original Design with Basic Optimizations (Synplify Pro)

In this experiment, the original RTL was synthesized in Synplify Pro using the default optimizations (without register retiming), then placed and routed in Xilinx ISE. The tool compared the output netlist to the RTL using combinational equivalence as in the previous section, and mapped inputs and outputs by name. It mapped 194 internal states by name and the remaining 40,985 states using proof-based methods. 360 EC determined all 41,177 states and 5 outputs to be equivalent. (Note the Synplify Pro implementation resulted in two more compare points than ISE). Total runtime was 2 minutes and 0 seconds.

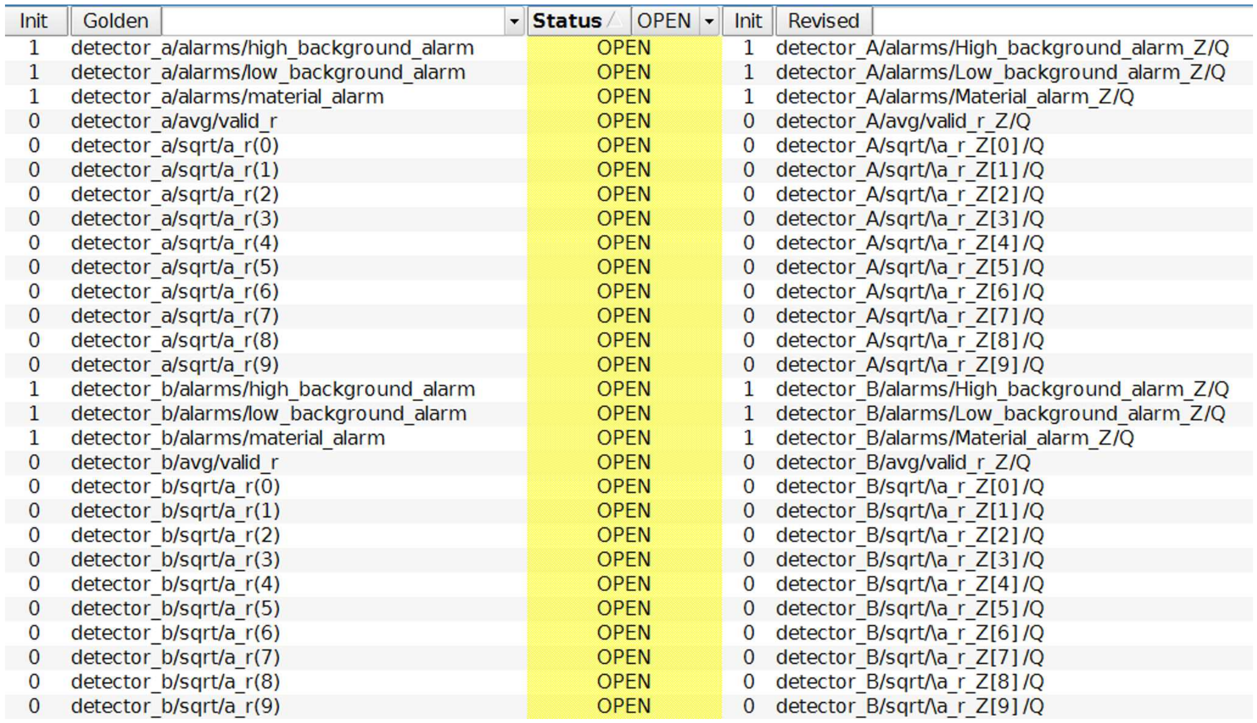
Command sequence:

```
map -nameonly
compute_state_relations
compare
```

Evaluation of Original Design with Retiming (Synplify Pro)

We repeated the previous experiment allowing Synplify Pro to perform register retiming optimizations. The tool mapped 158 states by name; however, proof-based methods were unable to map the additional 41,022 RTL states. With over 99% of the design unmapped, combinational equivalence had no basis for comparing the two designs, and the result was inconclusive. 80 compare points were combinationally equivalent (including the 5 outputs), 32 failed equivalence, and 51 were inconclusive.

We performed a subsequent, high-effort sequential equivalence step. After a total runtime of about one hour, the tool proved an additional 50 compare points sequentially equivalent. The following remaining 28 compare points remained inconclusive:



Init	Golden	Status OPEN	Init	Revised
1	detector_a/alarms/high_background_alarm	OPEN	1	detector_A/alarms/High_background_alarm_Z/Q
1	detector_a/alarms/low_background_alarm	OPEN	1	detector_A/alarms/Low_background_alarm_Z/Q
1	detector_a/alarms/material_alarm	OPEN	1	detector_A/alarms/Material_alarm_Z/Q
0	detector_a/avg/valid_r	OPEN	0	detector_A/avg/valid_r_Z/Q
0	detector_a/sqrt/a_r(0)	OPEN	0	detector_A/sqrt/\a_r_Z[0] /Q
0	detector_a/sqrt/a_r(1)	OPEN	0	detector_A/sqrt/\a_r_Z[1] /Q
0	detector_a/sqrt/a_r(2)	OPEN	0	detector_A/sqrt/\a_r_Z[2] /Q
0	detector_a/sqrt/a_r(3)	OPEN	0	detector_A/sqrt/\a_r_Z[3] /Q
0	detector_a/sqrt/a_r(4)	OPEN	0	detector_A/sqrt/\a_r_Z[4] /Q
0	detector_a/sqrt/a_r(5)	OPEN	0	detector_A/sqrt/\a_r_Z[5] /Q
0	detector_a/sqrt/a_r(6)	OPEN	0	detector_A/sqrt/\a_r_Z[6] /Q
0	detector_a/sqrt/a_r(7)	OPEN	0	detector_A/sqrt/\a_r_Z[7] /Q
0	detector_a/sqrt/a_r(8)	OPEN	0	detector_A/sqrt/\a_r_Z[8] /Q
0	detector_a/sqrt/a_r(9)	OPEN	0	detector_A/sqrt/\a_r_Z[9] /Q
1	detector_b/alarms/high_background_alarm	OPEN	1	detector_B/alarms/High_background_alarm_Z/Q
1	detector_b/alarms/low_background_alarm	OPEN	1	detector_B/alarms/Low_background_alarm_Z/Q
1	detector_b/alarms/material_alarm	OPEN	1	detector_B/alarms/Material_alarm_Z/Q
0	detector_b/avg/valid_r	OPEN	0	detector_B/avg/valid_r_Z/Q
0	detector_b/sqrt/a_r(0)	OPEN	0	detector_B/sqrt/\a_r_Z[0] /Q
0	detector_b/sqrt/a_r(1)	OPEN	0	detector_B/sqrt/\a_r_Z[1] /Q
0	detector_b/sqrt/a_r(2)	OPEN	0	detector_B/sqrt/\a_r_Z[2] /Q
0	detector_b/sqrt/a_r(3)	OPEN	0	detector_B/sqrt/\a_r_Z[3] /Q
0	detector_b/sqrt/a_r(4)	OPEN	0	detector_B/sqrt/\a_r_Z[4] /Q
0	detector_b/sqrt/a_r(5)	OPEN	0	detector_B/sqrt/\a_r_Z[5] /Q
0	detector_b/sqrt/a_r(6)	OPEN	0	detector_B/sqrt/\a_r_Z[6] /Q
0	detector_b/sqrt/a_r(7)	OPEN	0	detector_B/sqrt/\a_r_Z[7] /Q
0	detector_b/sqrt/a_r(8)	OPEN	0	detector_B/sqrt/\a_r_Z[8] /Q
0	detector_b/sqrt/a_r(9)	OPEN	0	detector_B/sqrt/\a_r_Z[9] /Q

Figure 31. Inconclusive Compare Points in Original Design with Register Retiming (Synplify)

Command sequence:

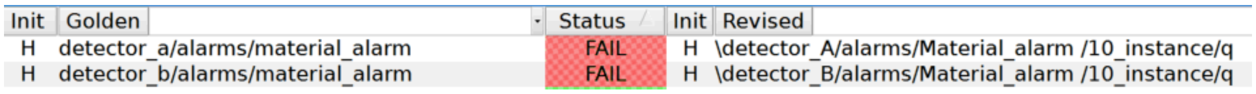
```
map -nameonly
compute_state_relations
compare
reset_compare_status -fail
compare -sequential
```

Evaluation of Modified Designs

Modified Design #1

For each modified design, a small but significant change was made to the RTL, which Xilinx ISE then synthesized, placed, and routed. We then compared the resulting netlist to the original (unmodified) RTL using the OneSpin 360 EC tool. Different individuals performed the RTL change and the subsequent verification, and the verifier was directed to avoid accessing the modified RTL until verification was complete.

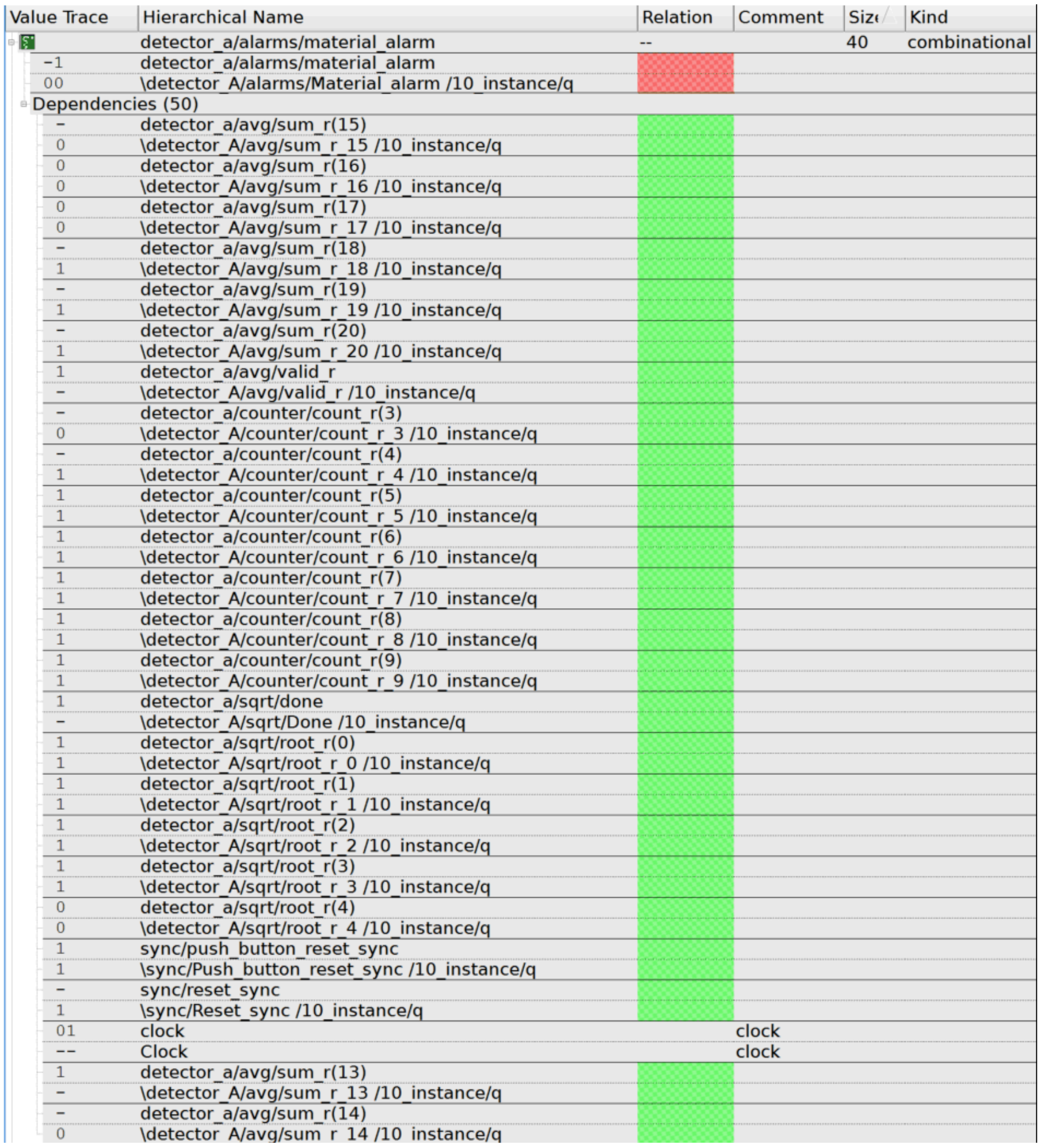
The first change was made to the sqrt.vhd file, modifying the square root function to add a constant value of 512 to the result. 360 EC found two compare points that failed combinationally due to the change. Sequential equivalence was inconclusive.



Init	Golden	Status	Init	Revised
H	detector_a/alarms/material_alarm	FAIL	H	\detector_A/alarms/Material_alarm /10_instance/q
H	detector_b/alarms/material_alarm	FAIL	H	\detector_B/alarms/Material_alarm /10_instance/q

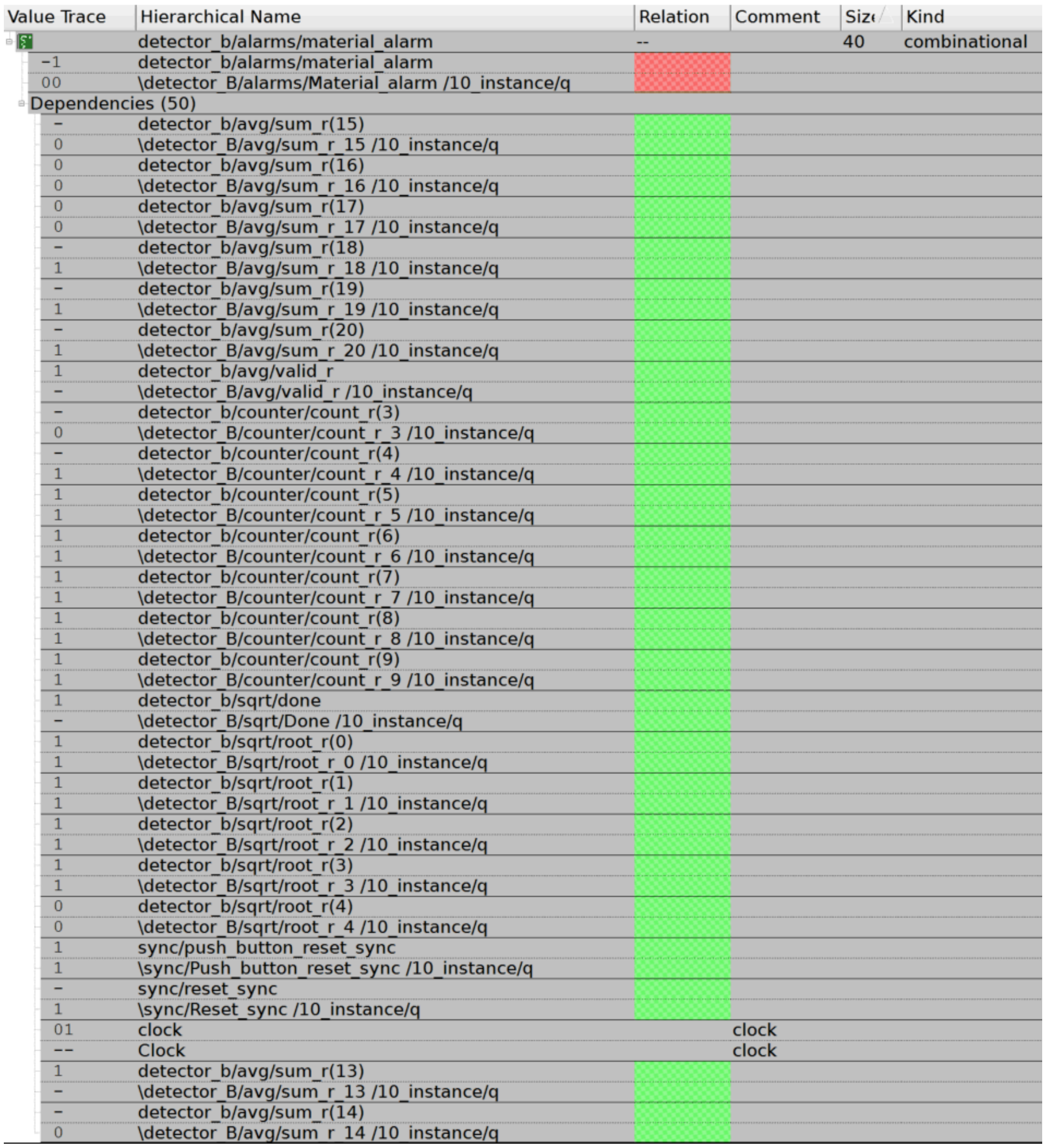
Figure 32. Failed Equivalence of Modified Design #1

The counterexample screens (shown below) indicates different logic on these states relative to sum_r, valid_r, count_r, and the output of the sqrt function.



Value Trace	Hierarchical Name	Relation	Comment	Size	Kind
	detector_a/alarms/material_alarm	--		40	combinational
-1	detector_a/alarms/material_alarm	🟥			
00	\detector_A/alarms/Material_alarm /10_instance/q	🟥			
Dependencies (50)					
–	detector_a/avg/sum_r(15)	🟩			
0	\detector_A/avg/sum_r_15 /10_instance/q	🟩			
0	detector_a/avg/sum_r(16)	🟩			
0	\detector_A/avg/sum_r_16 /10_instance/q	🟩			
0	detector_a/avg/sum_r(17)	🟩			
0	\detector_A/avg/sum_r_17 /10_instance/q	🟩			
–	detector_a/avg/sum_r(18)	🟩			
1	\detector_A/avg/sum_r_18 /10_instance/q	🟩			
–	detector_a/avg/sum_r(19)	🟩			
1	\detector_A/avg/sum_r_19 /10_instance/q	🟩			
–	detector_a/avg/sum_r(20)	🟩			
1	\detector_A/avg/sum_r_20 /10_instance/q	🟩			
1	detector_a/avg/valid_r	🟩			
–	\detector_A/avg/valid_r /10_instance/q	🟩			
–	detector_a/counter/count_r(3)	🟩			
0	\detector_A/counter/count_r_3 /10_instance/q	🟩			
–	detector_a/counter/count_r(4)	🟩			
1	\detector_A/counter/count_r_4 /10_instance/q	🟩			
1	detector_a/counter/count_r(5)	🟩			
1	\detector_A/counter/count_r_5 /10_instance/q	🟩			
1	detector_a/counter/count_r(6)	🟩			
1	\detector_A/counter/count_r_6 /10_instance/q	🟩			
1	detector_a/counter/count_r(7)	🟩			
1	\detector_A/counter/count_r_7 /10_instance/q	🟩			
1	detector_a/counter/count_r(8)	🟩			
1	\detector_A/counter/count_r_8 /10_instance/q	🟩			
1	detector_a/counter/count_r(9)	🟩			
1	\detector_A/counter/count_r_9 /10_instance/q	🟩			
1	detector_a/sqrt/done	🟩			
–	\detector_A/sqrt/Done /10_instance/q	🟩			
1	detector_a/sqrt/root_r(0)	🟩			
1	\detector_A/sqrt/root_r_0 /10_instance/q	🟩			
1	detector_a/sqrt/root_r(1)	🟩			
1	\detector_A/sqrt/root_r_1 /10_instance/q	🟩			
1	detector_a/sqrt/root_r(2)	🟩			
1	\detector_A/sqrt/root_r_2 /10_instance/q	🟩			
1	detector_a/sqrt/root_r(3)	🟩			
1	\detector_A/sqrt/root_r_3 /10_instance/q	🟩			
0	detector_a/sqrt/root_r(4)	🟩			
0	\detector_A/sqrt/root_r_4 /10_instance/q	🟩			
1	sync/push_button_reset_sync	🟩			
1	\sync/Push_button_reset_sync /10_instance/q	🟩			
–	sync/reset_sync	🟩			
1	\sync/Reset_sync /10_instance/q	🟩			
01	clock		clock		
––	Clock		clock		
1	detector_a/avg/sum_r(13)	🟩			
–	\detector_A/avg/sum_r_13 /10_instance/q	🟩			
–	detector_a/avg/sum_r(14)	🟩			
0	\detector_A/avg/sum_r_14 /10_instance/q	🟩			

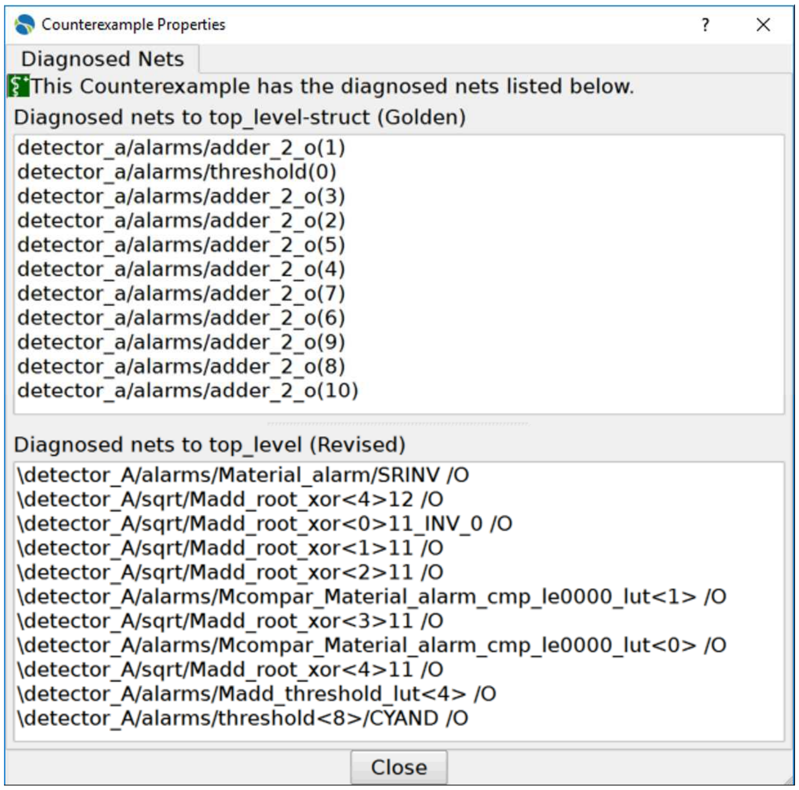
Figure 33. Counterexamples of Modified Design #1 (1)



Value Trace	Hierarchical Name	Relation	Comment	Siz	Kind
	detector_b/alarms/material_alarm	--		40	combinational
-1	detector_b/alarms/material_alarm	🟥			
00	\detector_B/alarms/Material_alarm /10_instance/q	🟥			
Dependencies (50)					
–	detector_b/avg/sum_r(15)	🟩			
0	\detector_B/avg/sum_r_15 /10_instance/q	🟩			
0	detector_b/avg/sum_r(16)	🟩			
0	\detector_B/avg/sum_r_16 /10_instance/q	🟩			
0	detector_b/avg/sum_r(17)	🟩			
0	\detector_B/avg/sum_r_17 /10_instance/q	🟩			
–	detector_b/avg/sum_r(18)	🟩			
1	\detector_B/avg/sum_r_18 /10_instance/q	🟩			
–	detector_b/avg/sum_r(19)	🟩			
1	\detector_B/avg/sum_r_19 /10_instance/q	🟩			
–	detector_b/avg/sum_r(20)	🟩			
1	\detector_B/avg/sum_r_20 /10_instance/q	🟩			
1	detector_b/avg/valid_r	🟩			
–	\detector_B/avg/valid_r /10_instance/q	🟩			
–	detector_b/counter/count_r(3)	🟩			
0	\detector_B/counter/count_r_3 /10_instance/q	🟩			
–	detector_b/counter/count_r(4)	🟩			
1	\detector_B/counter/count_r_4 /10_instance/q	🟩			
1	detector_b/counter/count_r(5)	🟩			
1	\detector_B/counter/count_r_5 /10_instance/q	🟩			
1	detector_b/counter/count_r(6)	🟩			
1	\detector_B/counter/count_r_6 /10_instance/q	🟩			
1	detector_b/counter/count_r(7)	🟩			
1	\detector_B/counter/count_r_7 /10_instance/q	🟩			
1	detector_b/counter/count_r(8)	🟩			
1	\detector_B/counter/count_r_8 /10_instance/q	🟩			
1	detector_b/counter/count_r(9)	🟩			
1	\detector_B/counter/count_r_9 /10_instance/q	🟩			
1	detector_b/sqrt/done	🟩			
–	\detector_B/sqrt/Done /10_instance/q	🟩			
1	detector_b/sqrt/root_r(0)	🟩			
1	\detector_B/sqrt/root_r_0 /10_instance/q	🟩			
1	detector_b/sqrt/root_r(1)	🟩			
1	\detector_B/sqrt/root_r_1 /10_instance/q	🟩			
1	detector_b/sqrt/root_r(2)	🟩			
1	\detector_B/sqrt/root_r_2 /10_instance/q	🟩			
1	detector_b/sqrt/root_r(3)	🟩			
1	\detector_B/sqrt/root_r_3 /10_instance/q	🟩			
0	detector_b/sqrt/root_r(4)	🟩			
0	\detector_B/sqrt/root_r_4 /10_instance/q	🟩			
1	sync/push_button_reset_sync	🟩			
1	\sync/Push_button_reset_sync /10_instance/q	🟩			
–	sync/reset_sync	🟩			
1	\sync/Reset_sync /10_instance/q	🟩			
01	clock		clock		
--	Clock		clock		
1	detector_b/avg/sum_r(13)	🟩			
–	\detector_B/avg/sum_r_13 /10_instance/q	🟩			
–	detector_b/avg/sum_r(14)	🟩			
0	\detector_B/avg/sum_r_14 /10_instance/q	🟩			

Figure 34. Counterexamples of Modified Design #1 (2)

360 EC's diagnosis of the counterexamples suggested some candidates for debugging:



This Counterexample has the diagnosed nets listed below.

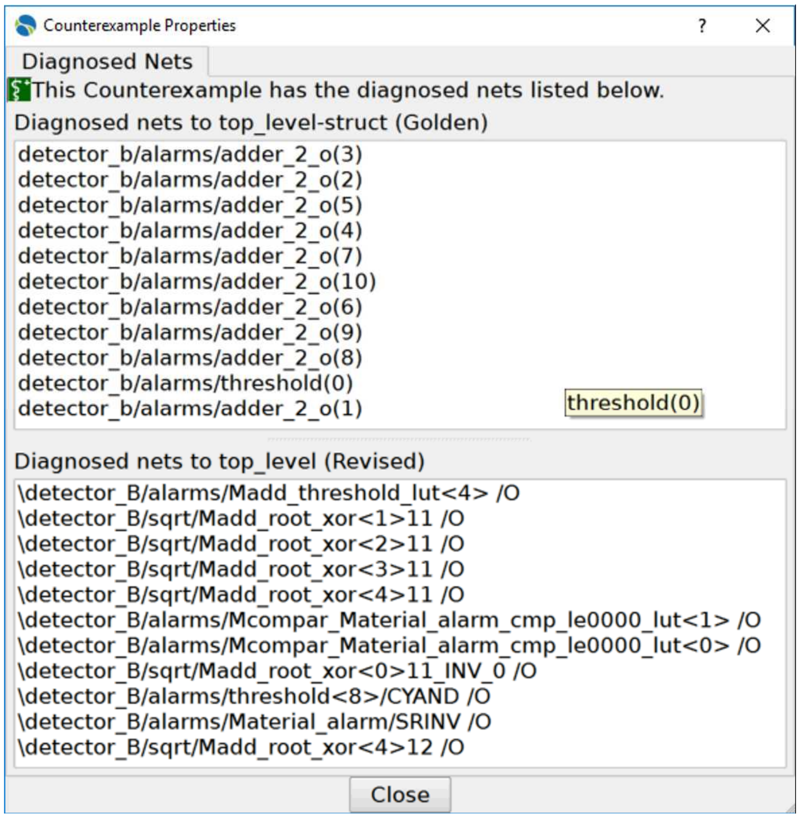
Diagnosed nets to top_level-struct (Golden)

```
detector_a/alarms/adder_2_o(1)
detector_a/alarms/threshold(0)
detector_a/alarms/adder_2_o(3)
detector_a/alarms/adder_2_o(2)
detector_a/alarms/adder_2_o(5)
detector_a/alarms/adder_2_o(4)
detector_a/alarms/adder_2_o(7)
detector_a/alarms/adder_2_o(6)
detector_a/alarms/adder_2_o(9)
detector_a/alarms/adder_2_o(8)
detector_a/alarms/adder_2_o(10)
```

Diagnosed nets to top_level (Revised)

```
\detector_A/alarms/Material_alarm/SRINV /O
\detector_A/sqrt/Madd_root_xor<4>12 /O
\detector_A/sqrt/Madd_root_xor<0>11_INV_0 /O
\detector_A/sqrt/Madd_root_xor<1>11 /O
\detector_A/sqrt/Madd_root_xor<2>11 /O
\detector_A/alarms/Mcompar_Material_alarm_cmp_le0000_lut<1> /O
\detector_A/sqrt/Madd_root_xor<3>11 /O
\detector_A/alarms/Mcompar_Material_alarm_cmp_le0000_lut<0> /O
\detector_A/sqrt/Madd_root_xor<4>11 /O
\detector_A/alarms/Madd_threshold_lut<4> /O
\detector_A/alarms/threshold<8>/CYAND /O
```

Figure 35. Candidate Differences of Modified Design #1 (1)



This Counterexample has the diagnosed nets listed below.

Diagnosed nets to top_level-struct (Golden)

```
detector_b/alarms/adder_2_o(3)
detector_b/alarms/adder_2_o(2)
detector_b/alarms/adder_2_o(5)
detector_b/alarms/adder_2_o(4)
detector_b/alarms/adder_2_o(7)
detector_b/alarms/adder_2_o(10)
detector_b/alarms/adder_2_o(6)
detector_b/alarms/adder_2_o(9)
detector_b/alarms/adder_2_o(8)
detector_b/alarms/threshold(0)
detector_b/alarms/adder_2_o(1)
```

Diagnosed nets to top_level (Revised)

```
\detector_B/alarms/Madd_threshold_lut<4> /O
\detector_B/sqrt/Madd_root_xor<1>11 /O
\detector_B/sqrt/Madd_root_xor<2>11 /O
\detector_B/sqrt/Madd_root_xor<3>11 /O
\detector_B/sqrt/Madd_root_xor<4>11 /O
\detector_B/alarms/Mcompar_Material_alarm_cmp_le0000_lut<1> /O
\detector_B/alarms/Mcompar_Material_alarm_cmp_le0000_lut<0> /O
\detector_B/sqrt/Madd_root_xor<0>11_INV_0 /O
\detector_B/alarms/threshold<8>/CYAND /O
\detector_B/alarms/Material_alarm/SRINV /O
\detector_B/sqrt/Madd_root_xor<4>12 /O
```

Figure 36. Candidate Differences of Modified Design #1 (2)

The diagnosed nets are related to the 'sigma' and 'average' busses. We determined there to be a change to one of these units but could not discern the exact functionality of the change without consulting the modified RTL. (The actual change was to the sqrt function which does indeed generate the sigma bus).

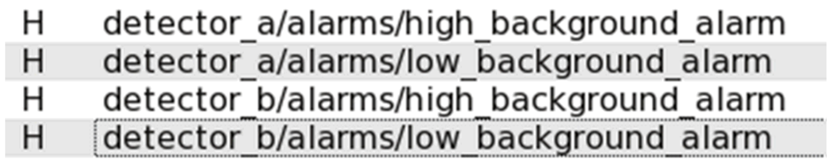
Command sequence:

```
map -input -output
map -nameonly
compute_state_relations
compare
diagnose_counterexamples -nores -golden -state
{{detector_a/alarms/material_alarm}}
diagnose_counterexamples -nores -golden -state
{{detector_b/alarms/material_alarm}}
```

Modified Design #2

The second modification altered detector.vhd, causing background alarms to never be set. This modification resulted in the removal of four associated registers from the synthesized netlist, as they were no longer used.

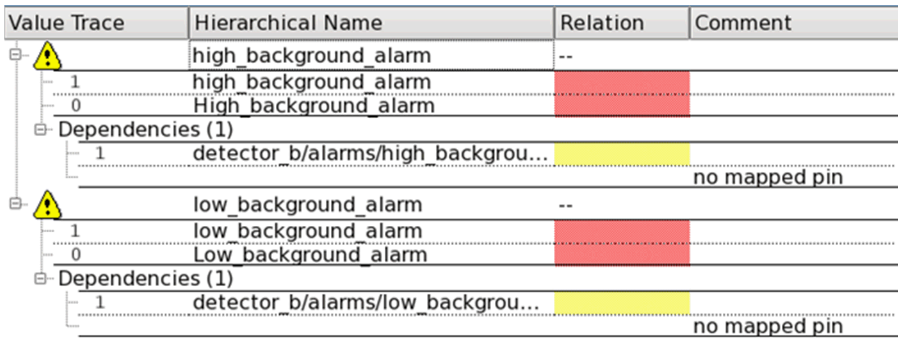
360 EC identified these registers in RTL as having no matching flip-flops in the netlist:



H	detector_a/alarms/high_background_alarm
H	detector_a/alarms/low_background_alarm
H	detector_b/alarms/high_background_alarm
H	detector_b/alarms/low_background_alarm

Figure 37. Removed Registers in Modified Design #2

As a result, combinational equivalence failed for both the High_background_alarm and Low_background_alarm outputs. 360 EC's diagnostic identified these failures as being related to the removed flip-flops:



Value Trace	Hierarchical Name	Relation	Comment
⚠	high_background_alarm	--	
1	high_background_alarm	🟥	
0	High_background_alarm	🟥	
Dependencies (1)			
1	detector_b/alarms/high_backgrou...	🟨	
			no mapped pin
⚠	low_background_alarm	--	
1	low_background_alarm	🟥	
0	Low_background_alarm	🟥	
Dependencies (1)			
1	detector_b/alarms/low_backgrou...	🟨	
			no mapped pin

Figure 38. Removed Flip-Flops in Modified Design #2

The comment "no mapped pin" means the RTL value of high_background_alarm and low_background_alarm each depend on the corresponding register inside the alarm_processor. These registers don't exist in the netlist.

In the netlist, it can be seen that High_background_alarm and Low_background_alarm have been shorted together and tied to ground:

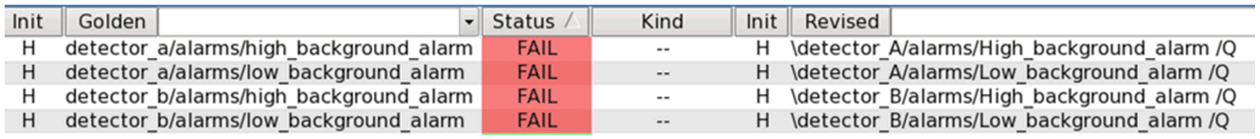
```
GND XST_GND (
.G(Low_background_alarm_OBUF_6)
);
...
OBUF High_background_alarm_OBUF (
.I(Low_background_alarm_OBUF_6),
.O(High_background_alarm)
);
OBUF Low_background_alarm_OBUF (
.I(Low_background_alarm_OBUF_6),
.O(Low_background_alarm)
);
```

Thus, we were able to determine the exact functionality of the change. Note that we do not claim to know which RTL file was changed, as the identical change could have been made to top_level.vhd, detector.vhd, or alarm_processor.vhd, all of which would have been indistinguishable. In this case, it happened to have been detector.vhd.

Modified Design #3

The third modification also disables the high and low alarms by changing the threshold values so that they can never be exceeded.

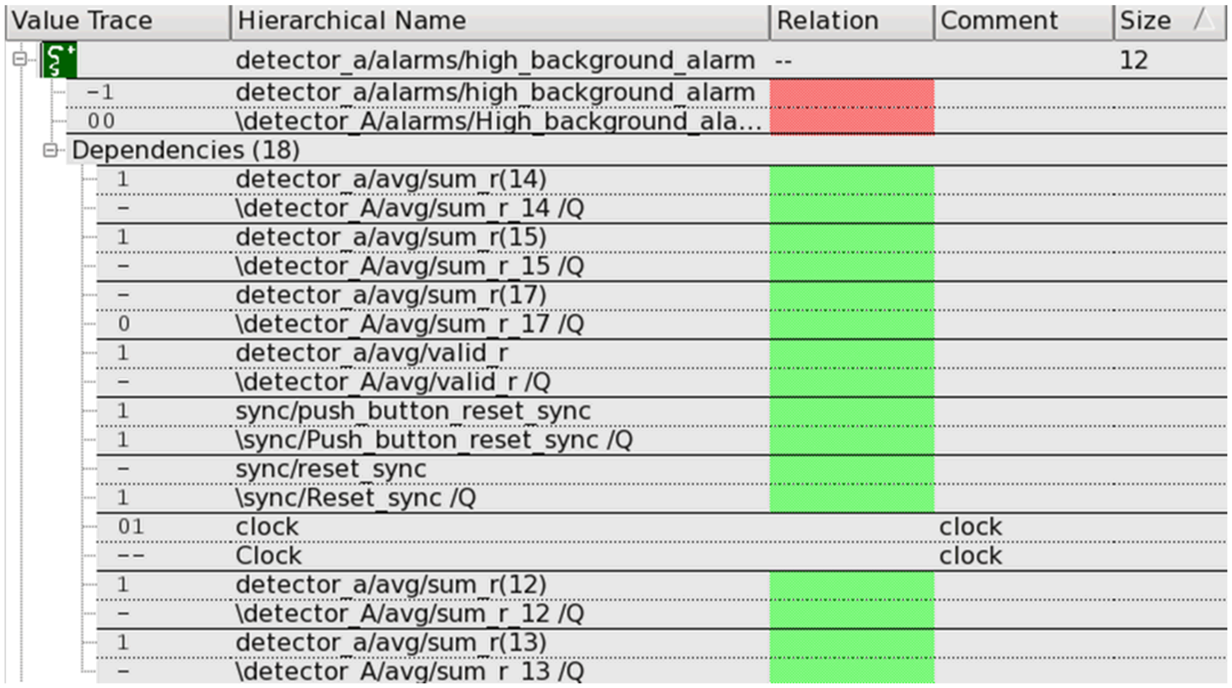
360 EC failed combinational equivalence on the following four equivalence points:



Init	Golden	Status	Kind	Init	Revised
H	detector_a/alarms/high_background_alarm	FAIL	--	H	\detector_A/alarms/High_background_alarm /Q
H	detector_a/alarms/low_background_alarm	FAIL	--	H	\detector_A/alarms/Low_background_alarm /Q
H	detector_b/alarms/high_background_alarm	FAIL	--	H	\detector_B/alarms/High_background_alarm /Q
H	detector_b/alarms/low_background_alarm	FAIL	--	H	\detector_B/alarms/Low_background_alarm /Q

Figure 39. Failed Equivalence of Modified Design #3

360 EC's diagnosis indicated potential disagreement in the threshold values used to set the alarm:



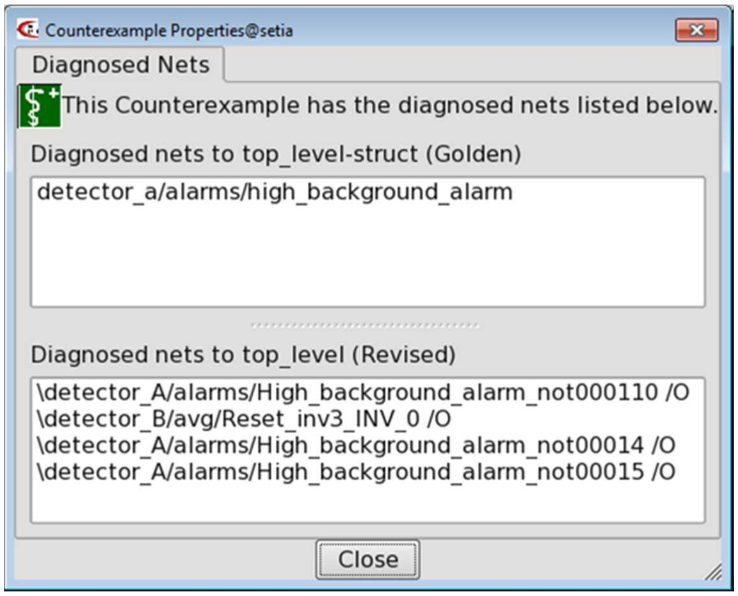
Value Trace	Hierarchical Name	Relation	Comment	Size
	detector_a/alarms/high_background_alarm	--		12
-1	detector_a/alarms/high_background_alarm	🟥		
00	\detector_A/alarms/High_background_ala...	🟥		
Dependencies (18)				
1	detector_a/avg/sum_r(14)	🟩		
–	\detector_A/avg/sum_r_14 /Q	🟩		
1	detector_a/avg/sum_r(15)	🟩		
–	\detector_A/avg/sum_r_15 /Q	🟩		
–	detector_a/avg/sum_r(17)	🟩		
0	\detector_A/avg/sum_r_17 /Q	🟩		
1	detector_a/avg/valid_r	🟩		
–	\detector_A/avg/valid_r /Q	🟩		
1	sync/push_button_reset_sync	🟩		
1	\sync/Push_button_reset_sync /Q	🟩		
–	sync/reset_sync	🟩		
1	\sync/Reset_sync /Q	🟩		
01	clock		clock	
--	Clock		clock	
1	detector_a/avg/sum_r(12)	🟩		
–	\detector_A/avg/sum_r_12 /Q	🟩		
1	detector_a/avg/sum_r(13)	🟩		
–	\detector_A/avg/sum_r_13 /Q	🟩		

Figure 40. Disagreement of Threshold Values in Modified Design #3

Initially it was thought the reset logic had been tampered with, but we found the flip-flop description in the netlist to be consistent with the RTL—and also the Material_alarm register, both of which the tool proved equivalent.

```
FDSE \detector_A/alarms/High_background_alarm (
.C(Clock_BUFGP_3),
.CE(\detector_A/alarms/High_background_alarm_not0001 ),
.D(\sync/Push_button_reset_sync_832 ),
.S(\detector_A/Reset_inv ),
.Q(\detector_A/alarms/High_background_alarm_34 )
);
FDSE \detector_A/alarms/Low_background_alarm (
.C(Clock_BUFGP_3),
.CE(\detector_A/alarms/Low_background_alarm_not0001 ),
.D(\sync/Push_button_reset_sync_832 ),
.S(\detector_A/Reset_inv ),
.Q(\detector_A/alarms/Low_background_alarm_39 )
);
FDSE \detector_A/alarms/Material_alarm (
.C(Clock_BUFGP_3),
.CE(\detector_A/alarms/Material_alarm_not0001 ),
.D(\sync/Push_button_reset_sync_832 ),
.S(\detector_A/Reset_inv ),
.Q(\detector_A/alarms/Material_alarm_61 )
);
```

Next, we examined the generating conditions for the high_background_alarm flip-flop and diagnosed them to a handful of nets:



This Counterexample has the diagnosed nets listed below.

Diagnosed nets to top_level-struct (Golden)

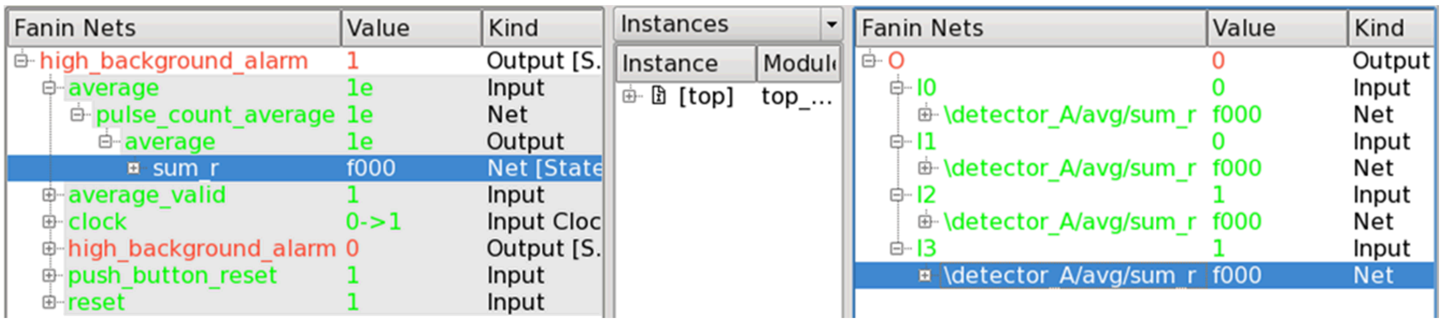
detector_a/alarms/high_background_alarm

Diagnosed nets to top_level (Revised)

\detector_A/alarms/High_background_alarm_not000110 /O
\detector_B/avg/Reset_inv3_INV_0 /O
\detector_A/alarms/High_background_alarm_not00014 /O
\detector_A/alarms/High_background_alarm_not00015 /O

Figure 41. Altered Nets in Modified Design #3

Plotting the diagnostic counterexample in a waveform revealed the RTL behavior the netlist failed to match: when sum_r reaches 0xF000, one of these signals should assert logic-high. However, none of them do, as indicated by the red highlight in the fanin net on the right hand side:



Fanin Nets	Value	Kind
high_background_alarm	1	Output [S.
average	1e	Input
pulse_count_average	1e	Net
average	1e	Output
sum_r	f000	Net [State
average_valid	1	Input
clock	0->1	Input Cloc
high_background_alarm	0	Output [S.
push_button_reset	1	Input
reset	1	Input

Instances

Instance	Modul
[top]	top_...

Fanin Nets	Value	Kind
O	0	Output
I0	0	Input
\detector_A/avg/sum_r	f000	Net
I1	0	Input
\detector_A/avg/sum_r	f000	Net
I2	1	Input
\detector_A/avg/sum_r	f000	Net
I3	1	Input
\detector_A/avg/sum_r	f000	Net

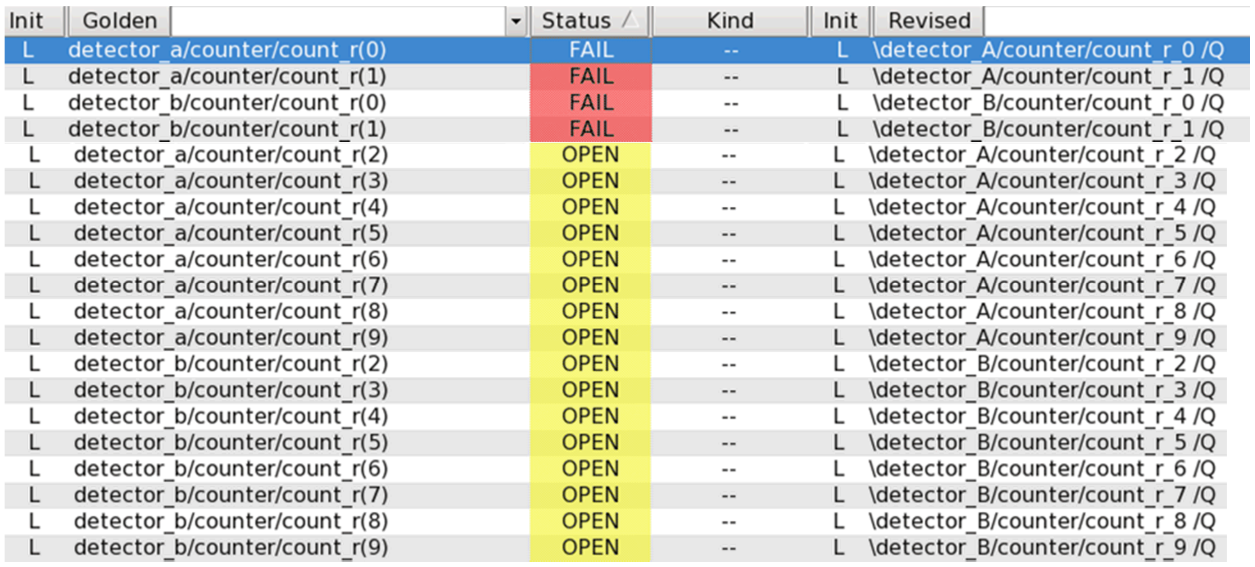
Figure 42. Signal Disagreement in Modified Design #3

Based on these observations, it was concluded that the values of HIGH_ALARM_LEVEL and LOW_ALARM_LEVEL were modified so that they no longer triggered at the appropriate point. This change was indeed the case in the modified RTL.

Modified Design #4

The fourth modification added logic to pulse_counter.vhd to only increment the counter on every other pulse, after the pulse count initially exceeds one-quarter its maximum value. This modification effectively inhibits or delays the alarm signals.

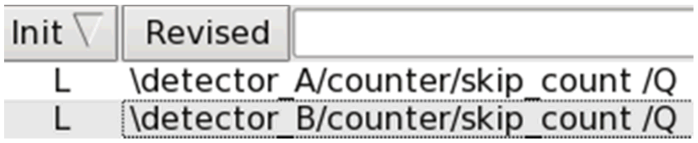
360 EC reported four failed equivalence points and 16 "open" points that could not be compared.



Init	Golden	Status	Kind	Init	Revised
L	detector_a/counter/count_r(0)	FAIL	--	L	\detector_A/counter/count_r_0 /Q
L	detector_a/counter/count_r(1)	FAIL	--	L	\detector_A/counter/count_r_1 /Q
L	detector_b/counter/count_r(0)	FAIL	--	L	\detector_B/counter/count_r_0 /Q
L	detector_b/counter/count_r(1)	FAIL	--	L	\detector_B/counter/count_r_1 /Q
L	detector_a/counter/count_r(2)	OPEN	--	L	\detector_A/counter/count_r_2 /Q
L	detector_a/counter/count_r(3)	OPEN	--	L	\detector_A/counter/count_r_3 /Q
L	detector_a/counter/count_r(4)	OPEN	--	L	\detector_A/counter/count_r_4 /Q
L	detector_a/counter/count_r(5)	OPEN	--	L	\detector_A/counter/count_r_5 /Q
L	detector_a/counter/count_r(6)	OPEN	--	L	\detector_A/counter/count_r_6 /Q
L	detector_a/counter/count_r(7)	OPEN	--	L	\detector_A/counter/count_r_7 /Q
L	detector_a/counter/count_r(8)	OPEN	--	L	\detector_A/counter/count_r_8 /Q
L	detector_a/counter/count_r(9)	OPEN	--	L	\detector_A/counter/count_r_9 /Q
L	detector_b/counter/count_r(2)	OPEN	--	L	\detector_B/counter/count_r_2 /Q
L	detector_b/counter/count_r(3)	OPEN	--	L	\detector_B/counter/count_r_3 /Q
L	detector_b/counter/count_r(4)	OPEN	--	L	\detector_B/counter/count_r_4 /Q
L	detector_b/counter/count_r(5)	OPEN	--	L	\detector_B/counter/count_r_5 /Q
L	detector_b/counter/count_r(6)	OPEN	--	L	\detector_B/counter/count_r_6 /Q
L	detector_b/counter/count_r(7)	OPEN	--	L	\detector_B/counter/count_r_7 /Q
L	detector_b/counter/count_r(8)	OPEN	--	L	\detector_B/counter/count_r_8 /Q
L	detector_b/counter/count_r(9)	OPEN	--	L	\detector_B/counter/count_r_9 /Q

Figure 43. Failed and Inconclusive Comparisons in Modified Design #4

The tool also reported two extra registers in the netlist which are not present in the RTL.

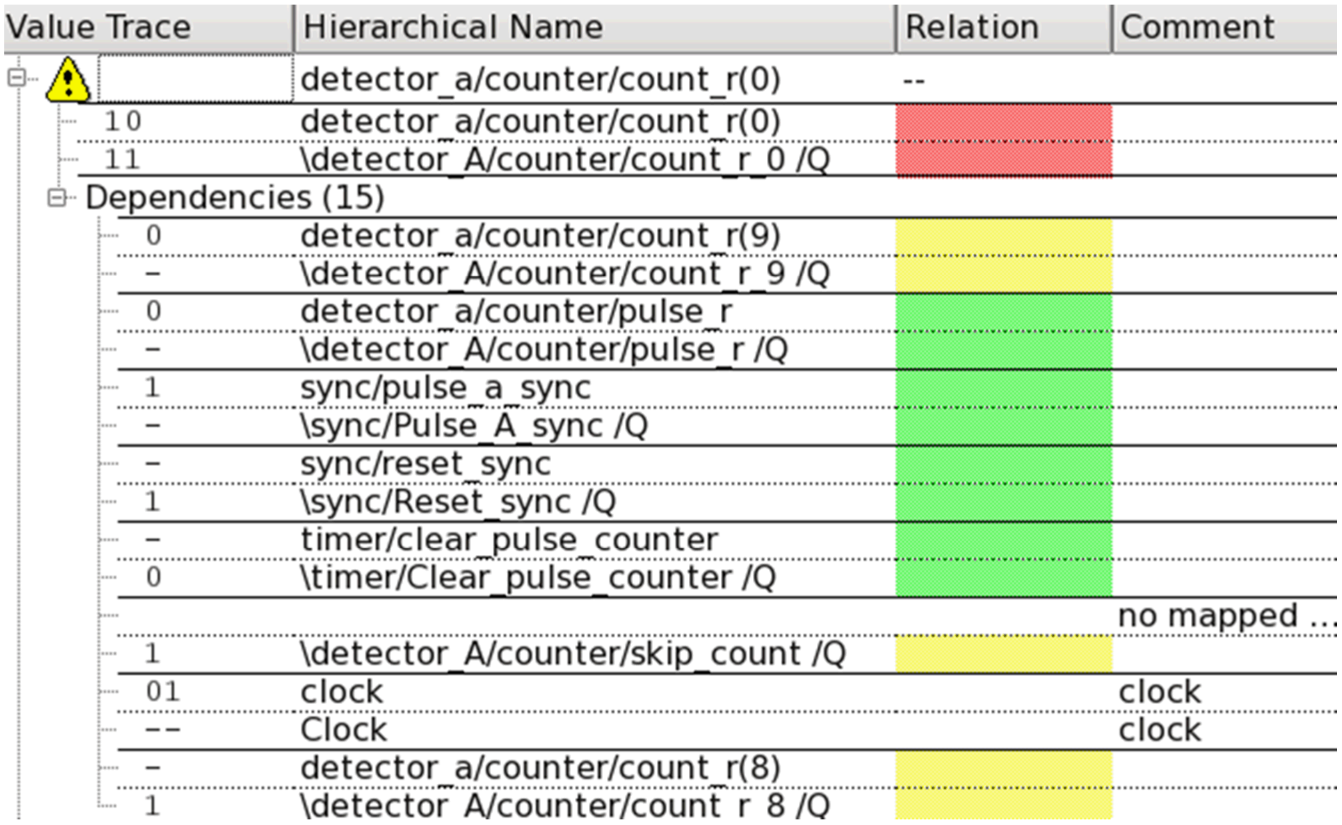


Init	Revised
L	\detector_A/counter/skip_count /Q
L	\detector_B/counter/skip_count /Q

Figure 44. Extra Registers in Modified Design #4

This additional logic is not by itself unusual, as it could indicate duplicated registers from optimization or a benign failure of synthesis to optimize away unused registers. However, there was no "skip_count" or related signal in the RTL, so we determined this extra logic to be an intentional modification of circuit functionality.

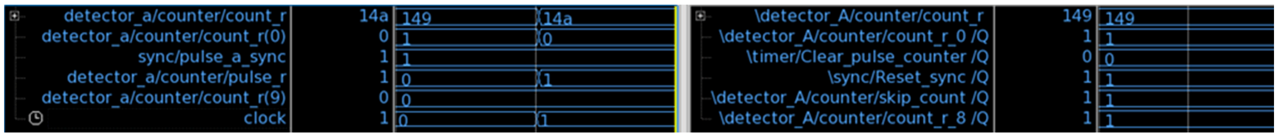
Indeed, one can see from the counterexample report that the next value of count_r(0) depends on this new skip_count register, which has no equivalent in the RTL.



Value Trace	Hierarchical Name	Relation	Comment
	detector_a/counter/count_r(0)	--	
10	detector_a/counter/count_r(0)	🟥	
11	\detector_A/counter/count_r_0 /Q	🟥	
Dependencies (15)			
0	detector_a/counter/count_r(9)	🟨	
–	\detector_A/counter/count_r_9 /Q	🟨	
0	detector_a/counter/pulse_r	🟩	
–	\detector_A/counter/pulse_r /Q	🟩	
1	sync/pulse_a_sync	🟩	
–	\sync/Pulse_A_sync /Q	🟩	
–	sync/reset_sync	🟩	
1	\sync/Reset_sync /Q	🟩	
–	timer/clear_pulse_counter	🟩	
0	\timer/Clear_pulse_counter /Q	🟩	
			no mapped ...
1	\detector_A/counter/skip_count /Q	🟨	
01	clock		clock
--	Clock		clock
–	detector_a/counter/count_r(8)	🟨	
1	\detector_A/counter/count_r_8 /Q	🟨	

Figure 45. Counterexamples in Modified Design #4

In the counterexample waveform, the left-hand circuit (representing the RTL) can be seen changing its count_r value from 0x149 to 0x14A, without a corresponding change in the right-hand circuit (representing the netlist). This lack of change is due to a value of '1' on skip_count.

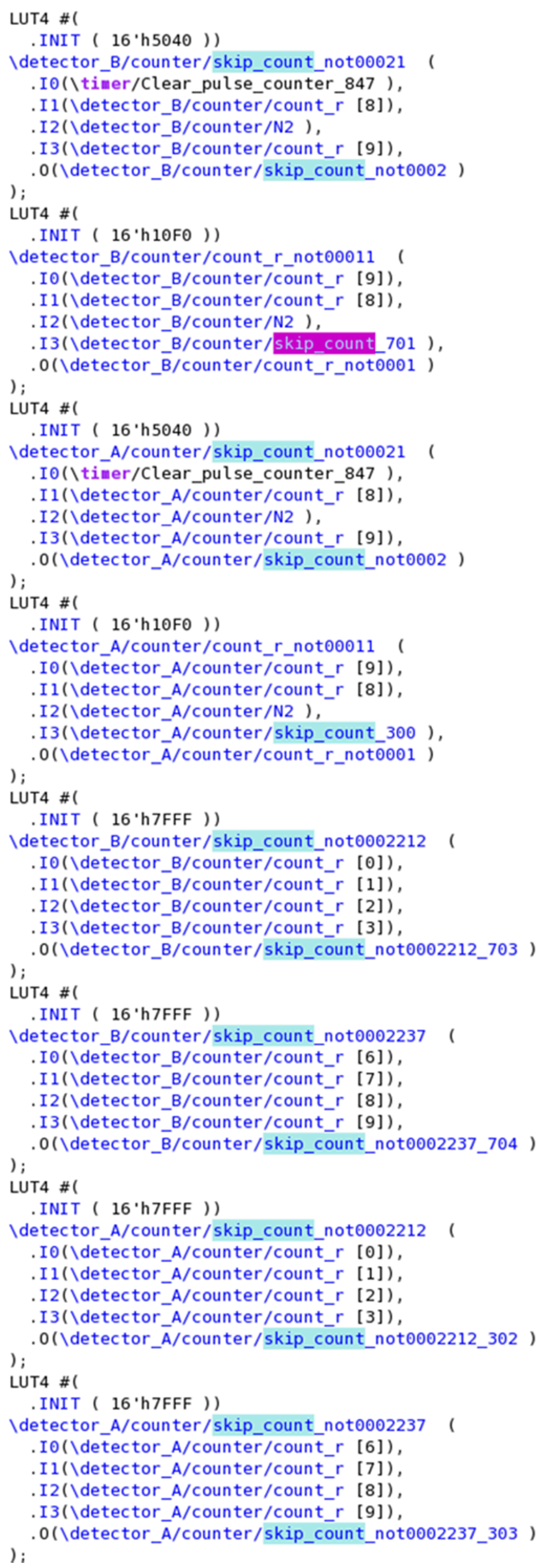


```
detector_a/counter/count_r       14a  149  14a     \detector_A/counter/count_r         149  149
detector_a/counter/count_r(0)      0  1    0       \detector_A/counter/count_r_0 /Q      1  1
sync/pulse_a_sync                  1  1            \timer/Clear_pulse_counter /Q         0  0
detector_a/counter/pulse_r         1  0    1       \sync/Reset_sync /Q                   1  1
detector_a/counter/count_r(9)      0  0            \detector_A/counter/skip_count /Q     1  1
clock                              1  0    1       \detector_A/counter/count_r_8 /Q      1  1
```

Figure 46. Counterexample Waveform for Modified Design #4 (1)

As discussed previously, in combinational equivalence the initial state S_0 is a free variable, and counterexamples are always two cycles long. Thus it cannot be determined from a single waveform *how* skip_count gets to be a '1', only that *if* it becomes '1', it subsequently inhibits count_r from incrementing.

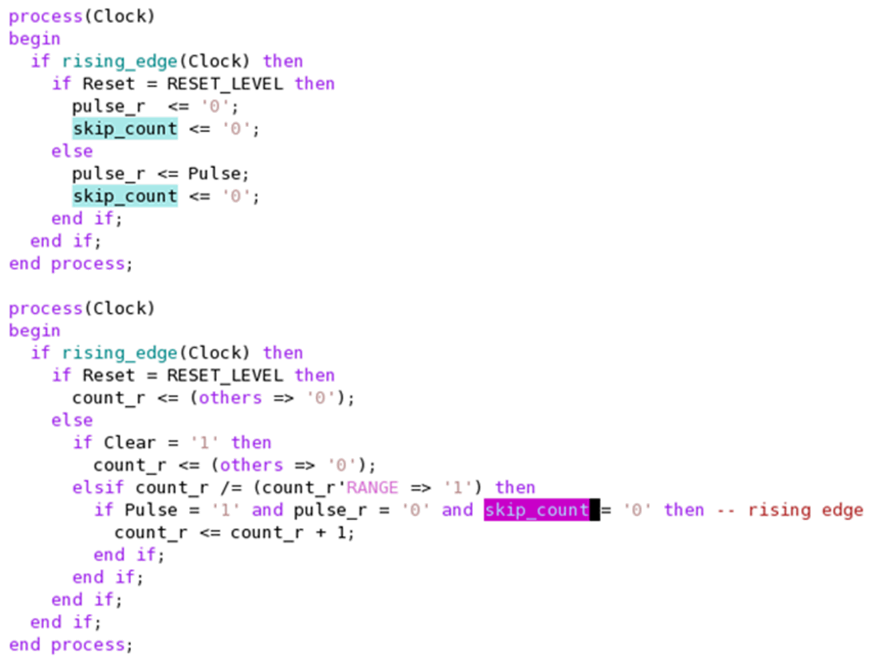
From the netlist once can see that skip_count is triggered by a certain count value but extracting that value from the netlist is not straightforward.



```
LUT4 #(
  .INIT ( 16'h5040 ))
\detector_B/counter/skip_count_not00021  (
  .I0(\timer/Clear_pulse_counter_847 ),
  .I1(\detector_B/counter/count_r [8]),
  .I2(\detector_B/counter/N2 ),
  .I3(\detector_B/counter/count_r [9]),
  .O(\detector_B/counter/skip_count_not0002 )
);
LUT4 #(
  .INIT ( 16'h10F0 ))
\detector_B/counter/count_r_not00011  (
  .I0(\detector_B/counter/count_r [9]),
  .I1(\detector_B/counter/count_r [8]),
  .I2(\detector_B/counter/N2 ),
  .I3(\detector_B/counter/skip_count_701 ),
  .O(\detector_B/counter/count_r_not0001 )
);
LUT4 #(
  .INIT ( 16'h5040 ))
\detector_A/counter/skip_count_not00021  (
  .I0(\timer/Clear_pulse_counter_847 ),
  .I1(\detector_A/counter/count_r [8]),
  .I2(\detector_A/counter/N2 ),
  .I3(\detector_A/counter/count_r [9]),
  .O(\detector_A/counter/skip_count_not0002 )
);
LUT4 #(
  .INIT ( 16'h10F0 ))
\detector_A/counter/count_r_not00011  (
  .I0(\detector_A/counter/count_r [9]),
  .I1(\detector_A/counter/count_r [8]),
  .I2(\detector_A/counter/N2 ),
  .I3(\detector_A/counter/skip_count_300 ),
  .O(\detector_A/counter/count_r_not0001 )
);
LUT4 #(
  .INIT ( 16'h7FFF ))
\detector_B/counter/skip_count_not0002212  (
  .I0(\detector_B/counter/count_r [0]),
  .I1(\detector_B/counter/count_r [1]),
  .I2(\detector_B/counter/count_r [2]),
  .I3(\detector_B/counter/count_r [3]),
  .O(\detector_B/counter/skip_count_not0002212_703 )
);
LUT4 #(
  .INIT ( 16'h7FFF ))
\detector_B/counter/skip_count_not0002237  (
  .I0(\detector_B/counter/count_r [6]),
  .I1(\detector_B/counter/count_r [7]),
  .I2(\detector_B/counter/count_r [8]),
  .I3(\detector_B/counter/count_r [9]),
  .O(\detector_B/counter/skip_count_not0002237_704 )
);
LUT4 #(
  .INIT ( 16'h7FFF ))
\detector_A/counter/skip_count_not0002212  (
  .I0(\detector_A/counter/count_r [0]),
  .I1(\detector_A/counter/count_r [1]),
  .I2(\detector_A/counter/count_r [2]),
  .I3(\detector_A/counter/count_r [3]),
  .O(\detector_A/counter/skip_count_not0002212_302 )
);
LUT4 #(
  .INIT ( 16'h7FFF ))
\detector_A/counter/skip_count_not0002237  (
  .I0(\detector_A/counter/count_r [6]),
  .I1(\detector_A/counter/count_r [7]),
  .I2(\detector_A/counter/count_r [8]),
  .I3(\detector_A/counter/count_r [9]),
  .O(\detector_A/counter/skip_count_not0002237_303 )
);
```

Figure 47. Netlist of Modified Design #4 showing *skip_count* Signal

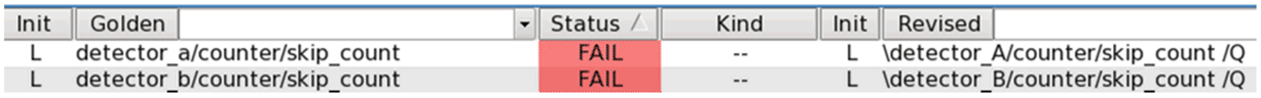
Therefore, to find this value, we added a constant skip_count register to a surrogate copy of the RTL, which was then compared to the netlist. This added register was given a constant value of ‘0’, which would cause 360 EC to produce a counterexample whenever the real skip_count changed value to ‘1’.



```
process(Clock)
begin
  if rising_edge(Clock) then
    if Reset = RESET_LEVEL then
      pulse_r  <= '0';
      skip_count <= '0';
    else
      pulse_r <= Pulse;
      skip_count <= '0';
    end if;
  end if;
end process;

process(Clock)
begin
  if rising_edge(Clock) then
    if Reset = RESET_LEVEL then
      count_r <= (others => '0');
    else
      if Clear = '1' then
        count_r <= (others => '0');
      elsif count_r /= (count_r'RANGE => '1') then
        if Pulse = '1' and pulse_r = '0' and skip_count = '0' then -- rising edge
          count_r <= count_r + 1;
        end if;
      end if;
    end if;
  end if;
end process;
```

Figure 48. Modified RTL to Confirm Failures Attributed to *skip_count*

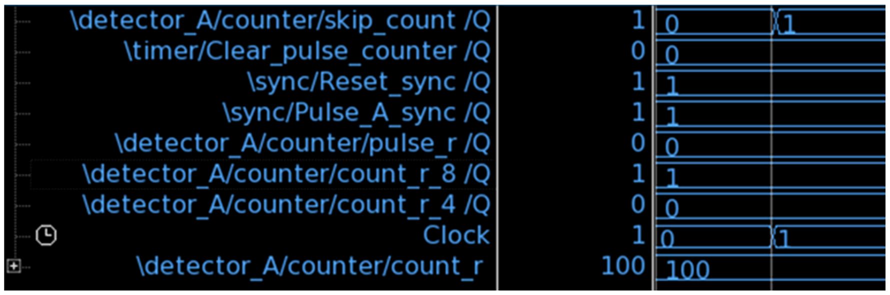
In the resulting comparison, the only failing equivalence points were the newly added skip_count registers.The count_r bits no longer failed equivalence, confirming the hypothesis that skip_count inhibits the counter from incrementing.



Init	Golden	Status	Kind	Init	Revised
L	detector_a/counter/skip_count	FAIL	--	L	\detector_A/counter/skip_count /Q
L	detector_b/counter/skip_count	FAIL	--	L	\detector_B/counter/skip_count /Q

Figure 49. Equivalence Failures in *skip_count* Test

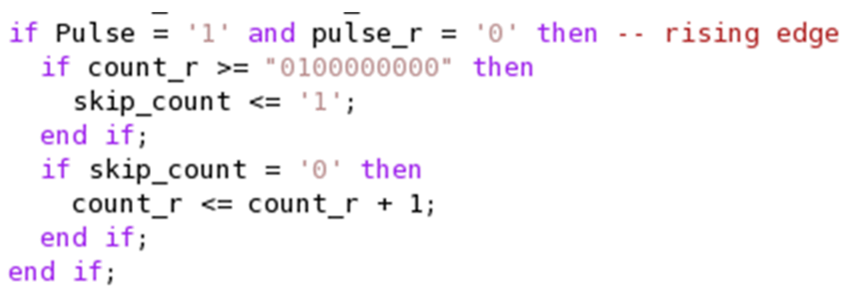
From the counterexample waveform, we determined that skip_count first asserts when count_r reaches a value of 100:



```
\detector_A/counter/skip_count /Q    1    0 | 1
\timer/Clear_pulse_counter /Q        0    0
\sync/Reset_sync /Q                  1    1
\sync/Pulse_A_sync /Q                1    1
\detector_A/counter/pulse_r /Q       0    0
\detector_A/counter/count_r_8 /Q     1    1
\detector_A/counter/count_r_4 /Q     0    0
Clock                                1    0 | 1
\detector_A/counter/count_r        100    100
```

Figure 50. Counterexample Waveform for Modified Design #4 (2)

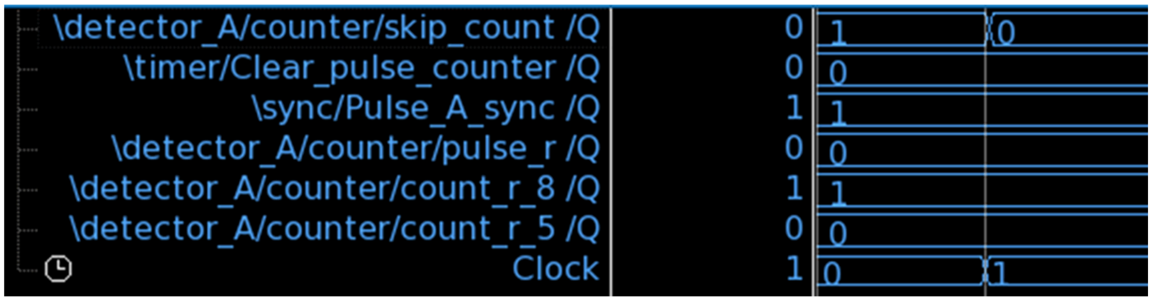
We then modified the surrogate copy of pulse_counter.vhd to set skip_count upon reaching this count value and repeated formal equivalence.



```
if Pulse = '1' and pulse_r = '0' then -- rising edge
  if count_r >= "0100000000" then
    skip_count <= '1';
  end if;
  if skip_count = '0' then
    count_r <= count_r + 1;
  end if;
end if;
```

Figure 51. Modified Surrogate Copy of *pulse_counter.vhd*

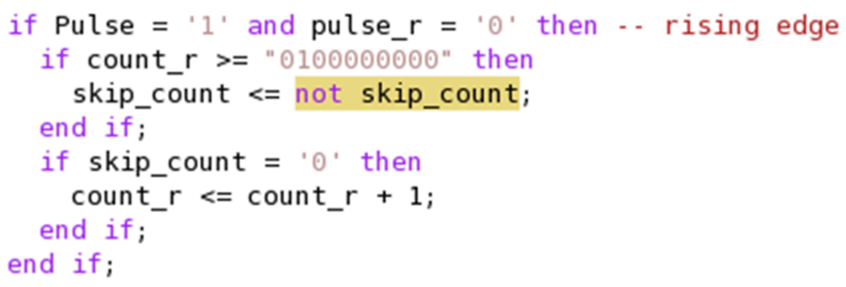
The resulting counterexample suggests skip_count is not a one-shot; it can go back to '0' after initially being set to '1'.



```
\detector_A/counter/skip_count /Q   0  1    0
\timer/Clear_pulse_counter /Q       0  0
\sync/Pulse_A_sync /Q               1  1
\detector_A/counter/pulse_r /Q      0  0
\detector_A/counter/count_r_8 /Q    1  1
\detector_A/counter/count_r_5 /Q    0  0
Clock                               1  0    1
```

Figure 52. Updated Counterexample Waveform

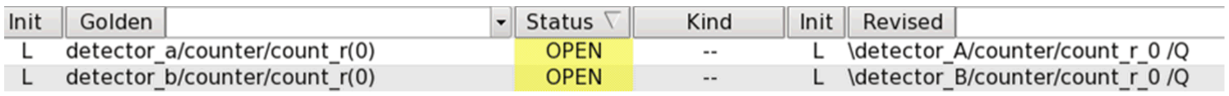
The following snippet represents the final guess at modified functionality; that is, skip_count toggles on every incoming pulse once initially asserted:



```
if Pulse = '1' and pulse_r = '0' then -- rising edge
  if count_r >= "0100000000" then
    skip_count <= not skip_count;
  end if;
  if skip_count = '0' then
    count_r <= count_r + 1;
  end if;
end if;
```

Figure 53. Final Modified Copy of *pulse_counter.vhd*

In the resulting EC, three bits of each counter fail combinationally; however, these count values passed sequential equivalence except for the respective least significant bits, which remained inconclusive:



Init	Golden	Status	Kind	Init	Revised
L	detector_a/counter/count_r(0)	OPEN	--	L	\detector_A/counter/count_r_0 /Q
L	detector_b/counter/count_r(0)	OPEN	--	L	\detector_B/counter/count_r_0 /Q

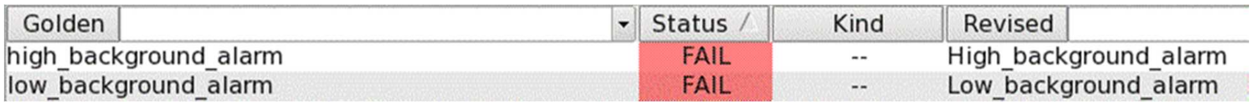
Figure 54. Remaining Inconclusive Bits

The actual RTL change was very similar to the guessed functionality but with slightly different timing conditions that caused these EC failures to remain.

Modified Design #5

The fifth and final modification replaced the OR operator, which combines background alarms from detector A and detector B, with an XOR operator, which always returns false whenever the detectors are in agreement. This change suppresses background alarms when both detectors see the same environment. A clever adversary could exploit this change by making the alarms appear to be working during a functional test in front of an inspector by setting off just one detector (with shielding or a hot source) instead of both.

360 EC reported the High_background_alarm and Low_backround_alarm outputs as non-equivalent. It reported all the internal states as equivalent.



Golden	Status	Kind	Revised
high_background_alarm	FAIL	--	High_background_alarm
low_background_alarm	FAIL	--	Low_background_alarm

Figure 55. Failed Equivalence in Modified Design #5

We therefore determined the change resided in the path between the alarm registers and the outputs. From the original RTL, an OR gate (or equivalent lookup table (LUT) function) was expected:

```
High_background_alarm <= high_background_alarm_A or high_background_alarm_B;
Low_background_alarm  <= low_background_alarm_A or low_background_alarm_B;
```

Figure 56. RTL Snippet Containing Background Alarm Logic

Instead, the schematic trace revealed a function, named Mxor, that outputs ‘0’ even though both alarm inputs are ‘1’.

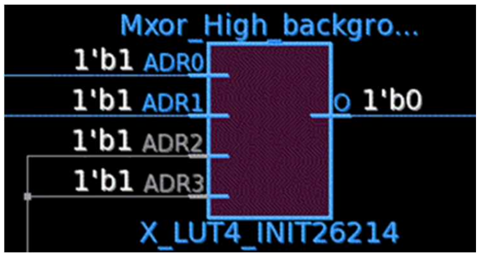


Figure 57. Modified Design #5 Schematic Trace Function Mxor

The Mxor name—and associated LUT configuration—suggested this modification replaced the expected OR functionality with an XOR gate. To confirm this suspicion, we created a surrogate copy of RTL with the suspected change and compared to the netlist using EC.

```
High_background_alarm <= high_background_alarm_A xor high_background_alarm_B;
Low_background_alarm  <= low_background_alarm_A xor low_background_alarm_B;
```

Figure 58. RTL Snippet with Surrogate Logic

360 EC proved the netlist equivalent to the surrogate RTL, confirming the inhibition of alarm signals due to the replacement of the OR gate.

Refutability

Formal equivalence is a powerful tool, but it has its limitations. It cannot prove the absence of malicious circuitry, only that any added or removed circuitry does not change the behavior of the digital model relative to its primary inputs and outputs. Other tools are required to verify the validity of the digital model in the presence of real-world delays, electromagnetic (EM) effects, and any unmodeled behavior of the FPGA logic elements and built-in circuitry.

Also, as noted above, the mapping process is sensitive to certain changes in the implementation, such as register optimization. Even a small number of unmapped states can cause a significant number of CEC failures and inconclusive SEC compare points, forcing a lengthy diagnostic process with no clear end.

This being said, to the extent the digital model can be said to faithfully represent real-world circuit behavior, the authors have been unable to produce any scenario in which EC falsely reported equivalence. The challenge remains largely in diagnosing and reversing false EC failures and inconclusive results.

4.6.3 Cost Estimation

360 EC is available in two flavors, EC-ASIC and EC-FPGA. EC-ASIC targets application specific integrated circuit (ASIC)-specific optimizations typical under Synopsys Design Compiler and costs around US$100,000 for a perpetual license with annual maintenance costs of tens of thousands of US dollars. EC-FPGA targets FPGA-specific applications typical with Synopsys Synplify and Xilinx ISE and costs a couple hundred thousand US dollars for a perpetual license with annual maintenance costs of tens of thousands of US dollars. The total labor to verify the original plus five modified designs (including preparation, debugging, and reporting) was 90 person hours of an expert in the tool.

4.7 Formal Equivalence Check with Formality

This style of formal equivalence checking is logically identical to Formal Equivalence Check using OneSpin but performed with a different tool. While OneSpin is an FPGA-specific tool, with FPGA-specific optimizations built-in, Formality is an ASIC-specific tool and has no *a priori* knowledge of FPGA architectures or naming heuristics. The use of Formality could thus be viewed as more trustable, as it makes no assumptions about the target hardware.

Formality was attempted early on, but dismissed as being unable to map the 40,000+ state bits from the memories. Black-boxing the memories was not an option for RTL-to-netlist comparison because the RTL did not instantiate memory elements directly but rather inferred them through register arrays. The inferred arrays would have to have been mapped independently to memory elements in a pseudo-synthesis. Formality (and black-boxing) could have been used in a netlist-to-netlist configuration, but this was not attempted. Alternatively, Formality could have been used to diagnose failing equivalence points after 360 EC performed the initial mapping step, but this option was not pursued in the interest of time. Formality has superior diagnostic capability but is significantly inferior in its ability to map and identify sequential optimizations.

4.8 Physical FPGA Authentication

There are various levels of tamper that could conceivably be performed on the physical FPGA parts. Here they are listed in an order of increasing cost and difficulty:

a. Silicon die replaced
b. Supplementary die added
c. Silicon die modified
d. Custom silicon fabricated to look similar to original

By depackaging an integrated circuit (IC) we would be reasonably certain to allow a. and b. to be seen by eye or under a microscope. X-ray imaging could assist in doing this in a non-destructive way but a party wanting higher confidence would need to resort to destructive analysis.

It is possible to make modifications to a silicon die (c.) using a focused ion beam (FIB). Within the semiconductor industry, FIB tools are commonly used for fault analysis and first circuit edits where the first spin of an IC is faulty but can be corrected to evaluate the design before going through a costly respin. These edits could feasibly be as small as 100 nm in size and would therefore be difficult to find by manual inspection.

Using altered original or reengineered photomasks it would be possible to make a silicon die that looked identical to the original (d.) but had modifications on the obscured lower layers. Detecting this would require not only depackaging but also delayering to allow examination of these lower layers.

Based on our previous research on simple microcontrollers, it has been shown that aluminium-based ICs can be successively delayered and imaged. This imaging would allow a party to compare integrated circuits from treaty equipment with their own supply. Due to the scale of the required imaging to cover all the circuitry of an IC, automated analysis is required to make comparisons a tractable problem. This comparison step has yet to be demonstrated.

There is a challenge applying our knowledge of physical authentication to FPGA ICs as they tend to be close to the technological cutting edge. There is a large element of churn with the older parts, which may be more easily authenticated, becoming obsolete and being replaced with faster and more complicated parts. There are a few exceptions to this; for instance, the Xilinx Coolrunner II and Spartan 3 range have been around for more than 10 years and show no signs of becoming obsolete. Figure 59 and Figure 60 show cross-sections of two of the FPGAs used to implement the basis system. These were chosen to be simple FPGAs but able to implement a useful amount of functionality. Table 5 summarises what was found and compares it to a simple microcontroller that has been used previously.

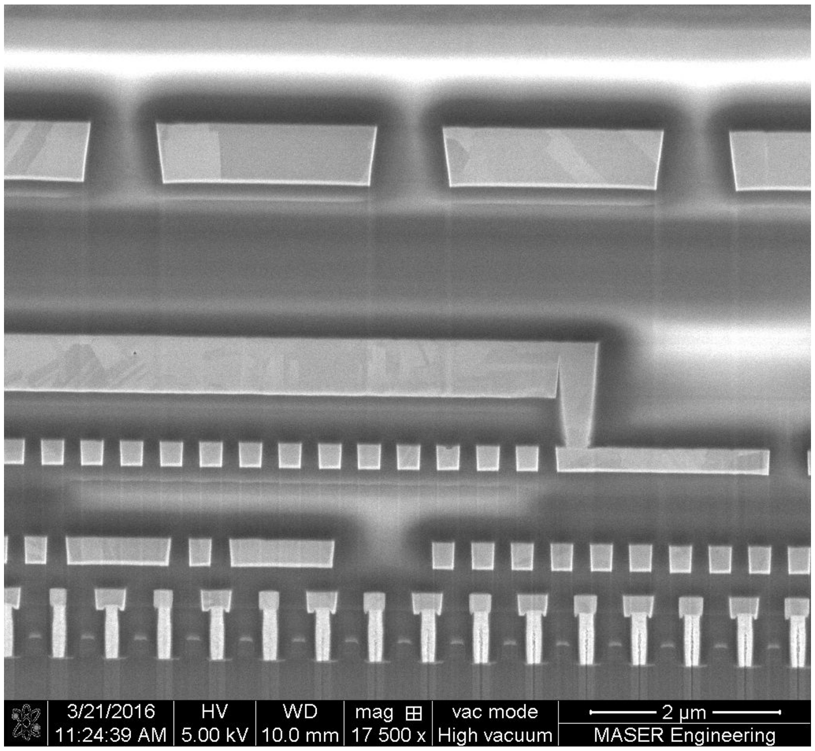


Figure 59. FIB cross-section of Microsemi ProAsic 3

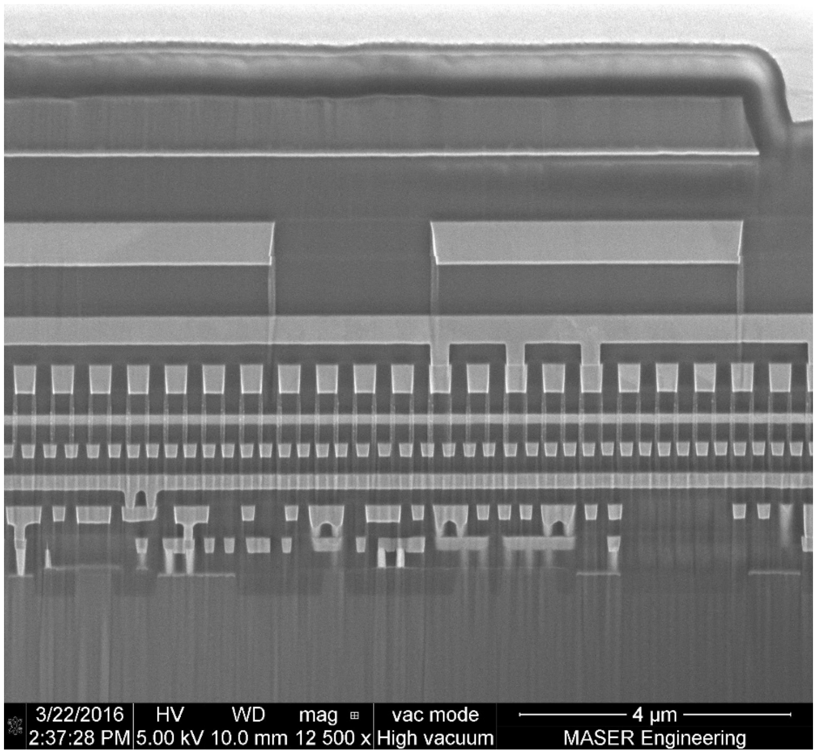


Figure 60. FIB cross-section of Xilinx Spartan 3E

Table 5. Imaging Summary of FPGAs

Part	**Layers**	**Technology node** (smallest feature size)	**Die size**	**Image size per layer** (estimate)
8-bit microcontroller ATMEGA328	3	180nm	3 x 3mm	2.5 GPixel
Microsemi ProASIC3 A3PN125-VQG100	8	130nm	2.5 x 3mm	4.2 GPixel
Xilinx Spartan 3E XC3S100E-TQG144	9	90nm	3 x 3mm	10 GPixel
Xilinx Coolrunner II XC2C64A	?	180nm	2 x 2.5mm	1.4 GPixel

Even though we have chosen older parts, they still use a small technology node with small transistor sizes and utilise many more interconnect layers with small vertical separation. The feature size is important as these parts are on the boundary of what can be optically resolved; using a scanning electron microscope (SEM) can overcome this issue but there are extra challenges therein. The large number of layers and the decreased vertical separation mean that revealing an entire layer, as has been done previously, may be impossible, but this is being investigated.

Additionally, these authentication activities could be complicated by the apparently common practise of subspec parts being labelled as a lower spec part by manufactures after disabling faulty features.

4.9 Bitstream Monitoring During Configuration (SRAM)

Bitstream monitoring during configuration of SRAM FPGAs (such as Xilinx) is a complementary technique to bitstream comparison of SRAM FPGAs, which is discussed in Section 4.10. Because the FPGA must receive bitstream data from the configuration memory over printed circuit board traces during power-on, an inspector can monitor and record this transfer of data for subsequent analysis. Since the inspector is connecting her own equipment to the electronics under inspection, the inspector must use a galvanically isolated interface, which allows information to flow out of the board, but not into it. This isolated interface assures the host that no malicious code is inserted during monitoring that could alter the FPGA configuration to provide false results or reveal sensitive information.

4.9.1 Theory of Operation

Because we designed the FPGA board to include the isolation electronics, all that we needed to monitor the bitstream data during configuration was a means to connect to this chip, to provide isolated power, and a way to sniff, or collect all serial information from, the serial peripheral interface (SPI) signals with which the configuration memory transfers configuration data. Figure 61 shows the portion of the FPGA board schematic containing the isolation electronics and connection interface.

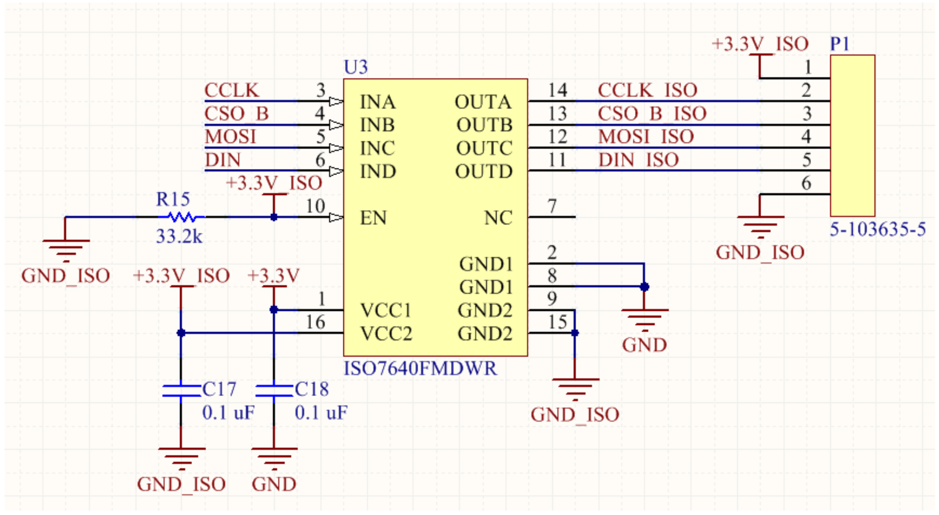


Figure 61. Isolation Hardware

The ISO764 (U3) provides galvanic isolation with a silicon dioxide capacitive insulation barrier. Figure 62 shows a conceptual schematic of the isolation circuit. Such a device is certified for 5 kV_{RMS} of galvanic isolation and also enforces one-way directionality of data transfer, making transfer in the opposite direction impossible by virtue of the circuit topology.

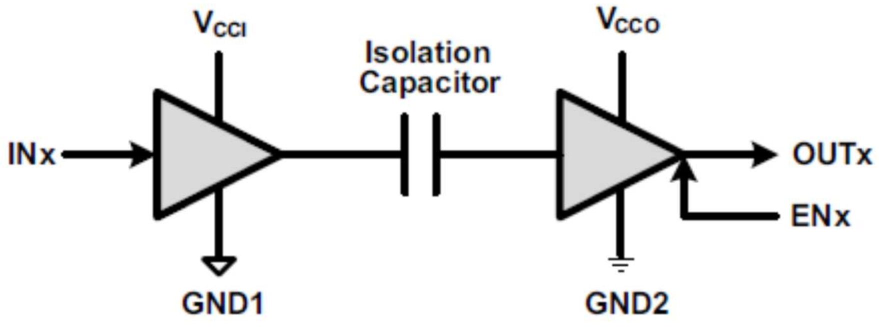


Figure 62. Simplified Isolation Schematic (courtesy Texas Instruments)

The four SPI signals – Chip Select (CSO_B), MOSI[14], DIN (MISO[15]), and Serial Clock (CCLK) – are brought to a flex cable connector (P1). Whatever device is attached to this connector must in turn provide isolated +3.3 V power and ground. While we could have designed a custom SPI sniffer, a quicker and easier solution was to use a commercial product, such as the Beagle SPI Sniffer[16]. In addition to sniffing SPI communication, this analyzer provides +5 V for powering small loads attached to it. As a result, we designed a small interposer board to accept the Beagle connector, transform its supplied voltage into a level acceptable to the isolation chip (+3.3 V), and provide the correct type of connector to directly attach to the FPGA board. Figure 63 shows the complete monitoring setup.

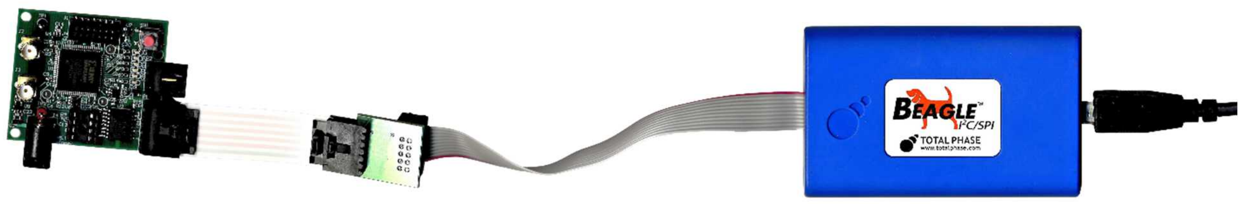


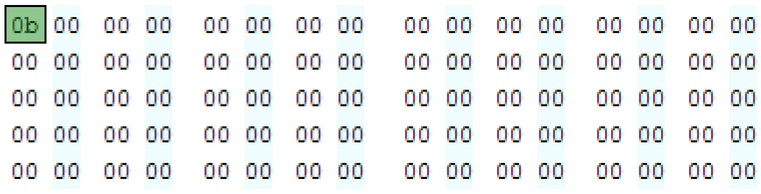
Figure 63. Complete SPI Monitoring Setup

When the FPGA powers on, it automatically begins the configuration sequence. However, the isolation electronics will also be powering on at the same time, resulting in temporary glitching on the isolated signals. Consequently, the SPI sniffer would frequently miss the falling edge of the chip select signal and not capture any data as a result. To mitigate this issue, we designed the FPGA board with a switch connected to the FPGA signal INIT_B. INIT_B is an FPGA input that the user can hold low to pause configuration. With this switch activated, the inspector could power on the board while holding configuration in a pause state. Once the board has fully powered on and the SPI sniffer is setup and ready to capture, the inspector can then release the switch and configuration will proceed. With this capability in place, the Beagle easily captures all configuration data.

Because SPI is a bidirectional interface, two streams of data exist for every configuration sequence. The first stream is the command data going from the FPGA to the configuration memory (MOSI). This

[14] MOSI stands for master out, slave in. SPI is a serial interface that has a defined master node and up to several slave nodes. The Chip Select is used to select from among the slave nodes for the serial data transmission.
[15] MISO stands for master in, slave out. This is the reverse direction of data transmission from MOSI.
[16] https://www.totalphase.com/products/beagle-i2cspi/

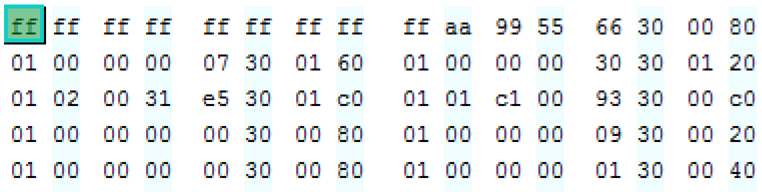
sequence begins with the Read Data Bytes at Higher Speed command, 0x0B. This command is the only one issued by the FPGA, so all subsequent traffic on this line should be 0x00.



```
0b 00  00 00  00 00  00 00  00 00  00 00  00 00  00 00
00 00  00 00  00 00  00 00  00 00  00 00  00 00  00 00
00 00  00 00  00 00  00 00  00 00  00 00  00 00  00 00
00 00  00 00  00 00  00 00  00 00  00 00  00 00  00 00
00 00  00 00  00 00  00 00  00 00  00 00  00 00  00 00
```

Figure 64. Initial Portion of Captured MOSI Data

The second data stream is the configuration data going from the configuration memory to the FPGA. This data will begin with a single byte of 0xFF during the command, a dummy word (0xFFFFFFFF), the synchronization word (0XAA995566), and then the rest of the configuration data.



```
ff ff  ff ff  ff ff  ff ff  ff aa  99 55  66 30  00 80
01 00  00 00  07 30  01 60  01 00  00 00  30 30  01 20
01 02  00 31  e5 30  01 c0  01 01  c1 00  93 30  00 c0
01 00  00 00  00 30  00 80  01 00  00 00  09 30  00 20
01 00  00 00  00 30  00 80  01 00  00 00  01 30  00 40
```

Figure 65. Initial Portion of Captured MISO Data

For this technique, we captured both MISO and MOSI streams for the golden design, as well as each buggy design. For the MOSI data, we checked that the data started with 0x0B and that every subsequent byte was 0x00. Passing this test indicates that no other commands were issued by the FPGA to the configuration memory besides the Read command. For the MISO data, we used custom Java software that redacts the captured data before the synchronization word and after the desynchronization word and then compares what is left to a trusted reference. The software displays any discrepancies and then reports the total number of mismatches at the end.

4.9.2 Performance Results

We have confirmed that bitstream monitoring during configuration is a useable technique to verify that the actual bitstream data loaded by an FPGA during configuration matches a pre-known trusted reference, provided that the inspector or host can delay or restart configuration once the system voltage has stabilized.

For this technique, it is important to also monitor the data going from the FPGA to the configuration memory. This data should only contain a read command. Any other commands, such as a write command to change the memory contents, are an immediate red flag.

This technique's run time will scale linearly with the amount of captured configuration data.

Evaluation of Original Design

For this evaluation, we captured the golden bitstream (with compression turned on) and compared it versus its unmodified self. We recorded 62,266 data bytes and found zero mismatches. We also captured the data transmitted from the FPGA to the memory. It matched the expected pattern (0x0B followed by all 0x00) and was an identical size.

Evaluation of Modified Designs

We recorded and analyzed all five modified bitstreams, and analyzed them for any differences in the size or content. The results are displayed in Table 6.

Source Bitstream	Captured Size (bytes)	# of Mismatches in Monitored Bitstream
Trusted	62,266	0
Modified #1	61,598	11,419
Modified #2	62,658	10,863
Modified #3	61,018	11,056
Modified #4	59,694	11,007
Modified #5	59,270	10,725

Table 6. Bitstream Monitoring During Configuration Comparison Results

Every modified bitstream was immediately identifiable as altered. The size of each bitstream differed from the trusted by 1-5%, while the number of mismatches was always over 10,000. For modified bitstreams that are larger than the trusted, the number of mismatches is identical to those found in the bitstream comparison of SRAM FPGAs, as seen in Section 4.10. For modified bitstreams that are smaller than the golden, the discrepancy in the number of mismatches is 4. The difference is due to the fact that the FPGA does not load the entire desynchronization pattern. It only loads the DESYNC command (0x0000000D) and not the four subsequent NOP commands (0x20000000).

While we also captured data going from the FPGA to the configuration memory for each of these tests, it passed every time as we have no easy way for the FPGA to issue illicit commands.

Refutability

While this technique seems straightforward, several issues exist. First, the inspector and host must have confidence in the isolation electronics. The host may have certification concerns and require that a physical air gap be present, requiring the use of optical isolation. Likewise, the inspector must be assured that the parts chosen do not redact or modify the transmitted bitstream data in any way through authentication of those parts or of the built board as a whole.

Further, the inspector would also want to monitor for out of band communication. The chosen sniffer expects data conforming to the SPI protocol, with a maximum speed of 50 MHz, and sampling only on the rising edge. Communication could be hidden by operating outside these limits. For example, data could be transmitted out of the configuration memory on both the rising and the falling edge of the serial clock. The proper bitstream data could be transmitted on the rising edge and ignored, while the subverted bitstream could be transmitted on the falling edge and used. This falling edge data would potentially never be seen by the SPI sniffer, though it may register it as an error condition.

Another concern is that the transmitted bitstream data itself might be planted to pass this technique. The FPGA could have a hidden configuration memory built in to it and would simply configure from it while receiving, but ignoring, all data from the external configuration memory. Such a scenario would be detectable via physical authentication of the FPGA itself.

4.9.3 Cost Estimation

The equipment for this method exists and would cost less than $200 per unit to construct more. No additional costs are necessary, unless more robust monitoring is desired, and the labor involved to run comparisons is trivial.

4.10 Bitstream Comparison (SRAM)

Bitstream comparison is a technique to authenticate the contents of the non-volatile memory from which a SRAM-based FPGA configures. Such FPGAs are volatile; the configuration data is stored internally in an array of latches. Because the FPGA cannot maintain data without a power source, it must be configured upon boot. This configuration data is in turn stored in a non-volatile memory, typically flash-based, which is co-located with the FPGA. The data is transferred on power-up via a serial or parallel link to the FPGA.

By socketing this configuration memory, rather than directly soldering it to the board, the inspector can easily remove the memory post-inspection and perform analysis on it. In this case, the analysis is a readout of the contents of the device and a comparison to a golden, or trusted, copy.

4.10.1 Theory of Operation

To read the contents of the non-volatile memory containing the bitstream, we designed a custom USB-based board, shown in Figure 66. This board accepts the socketed flash memory and contains a microcontroller with custom firmware to interrogate its contents.

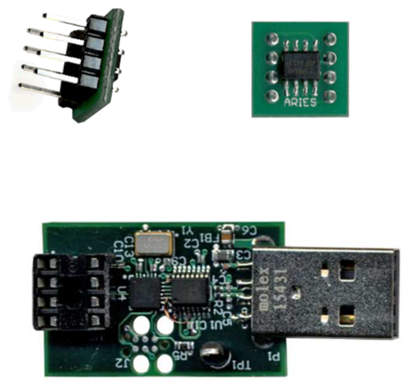
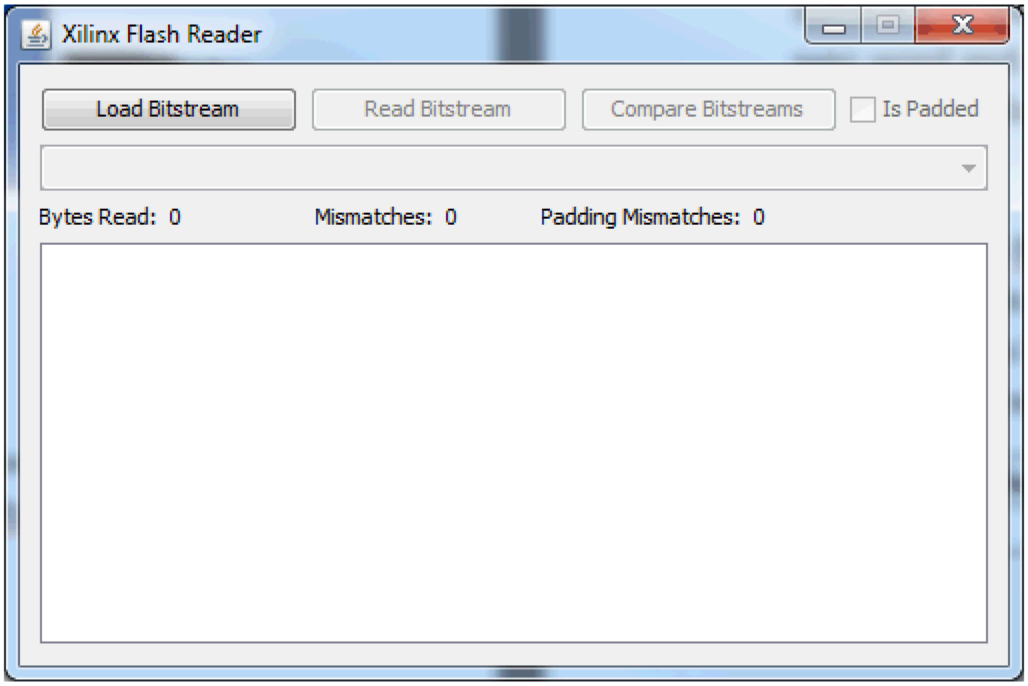


Figure 66. Socketed Memory and USB-based Reader

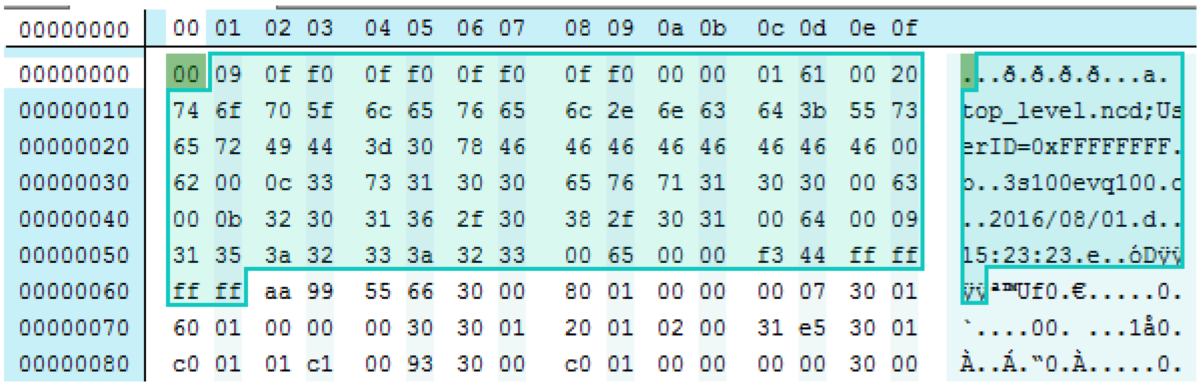
After inserting the flash memory into the socket, the board is plugged into an inspector’s computer where a custom program to reads the bitstream and compares it to a trusted reference. This program launches a graphical user interface (GUI) as shown in Figure 67.



Bytes Read: 0 Mismatches: 0 Padding Mismatches: 0

Figure 67. Java-based Bitstream Interrogation GUI

As a first step, the inspector loads the trusted bitstream. This bitstream represents a trustable copy of what the interrogated FPGA bitstream should contain. Any time a bitstream is loaded, the software analyzes it to ensure that it is valid. Specifically, all Xilinx bitstreams contain a synchronization word (0xAA995566) that the FPGA detects to begin the configuration process. Any data prior to this synchronization word is ignored during configuration. Such data includes the name of the top level module in the design, the date the bitstream was generated, the targeted FPGA, and other miscellaneous header information. While this data is present in the bitstream generated during application development, it is not actually downloaded into the flash memory during device programming. Figure 68 displays a typical Xilinx bitstream header.

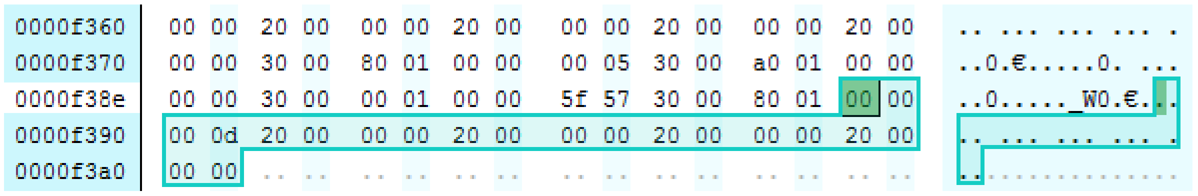


```
00000000  00 01 02 03  04 05 06 07  08 09 0a 0b  0c 0d 0e 0f

00000000  00 09 0f f0  0f f0 0f f0  0f f0 00 00  01 61 00 20   ...ð.ð.ð.ð...a.
00000010  74 6f 70 5f  6c 65 76 65  6c 2e 6e 63  64 3b 55 73   top_level.ncd;Us
00000020  65 72 49 44  3d 30 78 46  46 46 46 46  46 46 46 00   erID=0xFFFFFFFF.
00000030  62 00 0c 33  73 31 30 30  65 76 71 31  30 30 00 63   b..3s100evq100.c
00000040  00 0b 32 30  31 36 2f 30  38 2f 30 31  00 64 00 09   ..2016/08/01.d..
00000050  31 35 3a 32  33 3a 32 33  00 65 00 00  f3 44 ff ff   15:23:23.e..óDÿÿ
00000060  ff ff aa 99  55 66 30 00  80 01 00 00  00 07 30 01   ÿÿª™Uf0.€.....0.
00000070  60 01 00 00  00 30 30 01  20 01 02 00  31 e5 30 01   `....00. ...1å0.
00000080  c0 01 01 c1  00 93 30 00  c0 01 00 00  00 00 30 00   À..Á."0.À.....0.
```

Figure 68. Xilinx Bitstream Header

The software looks to see that this synchronization pattern is present in the loaded bitfile. If so, it strips off all data prior to this pattern and reports the number of redacted bytes to the inspector. If not, the inspector is warned that the bitstream is invalid.

Similarly, each bitstream contains a desynchronization sequence (0x0000000D followed by 0x20000000 repeated four times). This pattern indicates the end of a Xilinx bitstream, and any data afterwards is similarly ignored by the FPGA. The Java software also ensures that this pattern is present at the end of the file and then redacts any subsequent data. The software also alerts the inspector that superfluous data may be present. Figure 71 shows the ending of a typical Xilinx bitstream.

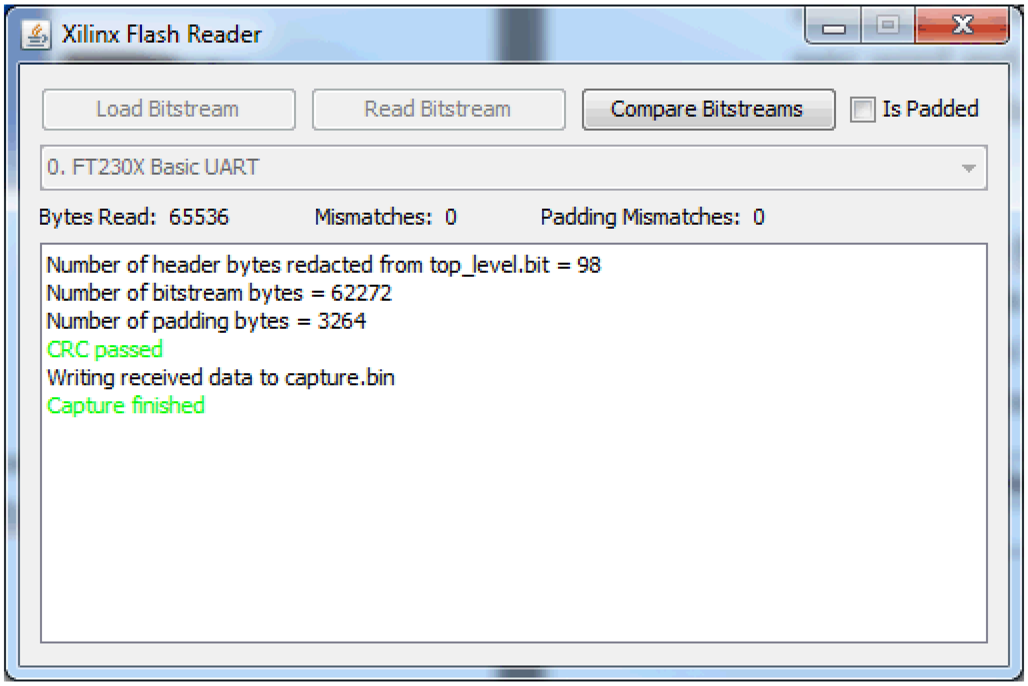


```
0000f360   00 00 20 00  00 00 20 00  00 00 20 00  00 00 20 00   .. ... ... ... .
0000f370   00 00 30 00  80 01 00 00  00 05 30 00  a0 01 00 00   ..0.€.....0. ...
0000f38e   00 00 30 00  00 01 00 00  5f 57 30 00  80 01 00 00   ..0....._W0.€...
0000f390   00 0d 20 00  00 00 20 00  00 00 20 00  00 00 20 00   .. ... ... ... .
0000f3a0   00 00 .. ..  .. .. .. ..  .. .. .. ..  .. .. .. ..   ................
```

Figure 69. Xilinx Bitstream Desync Pattern

Next, the inspector inserts the socketed flash memory under inspection and reads its contents. A command is sent by the software to the microcontroller on the USB board to begin the data interrogation process. The microcontroller responds by first querying the flash memory and determining its size. This capacity is sent to the computer followed by the entire contents of the flash memory, byte by byte. Because the bitstream may not fill the entire capacity of the flash memory, some number of padding bytes will exist. These bytes should contain all '1's (0xFF). The Java software knows how many bytes are in the trusted bitstream and can thus calculate the number of padding bytes present in the flash memory, i.e., the memory capacity - the number of bitstream bytes = the number of padding bytes. This number is reported to the GUI.

Also, during transmission, a cyclic redundancy check (CRC) of the sent data is calculated to ensure reliable delivery. Once data transmission is complete, the resultant CRC is communicated to the software. The software will then compare this CRC to its own calculated CRC and check to see if they match, the result of which is shown to the user. If the CRC passes, the software will write the captured bitstream data to a file and alert the inspector. If not, the inspector will be notified and should retry the process from the beginning.



Bytes Read: 65536 Mismatches: 0 Padding Mismatches: 0

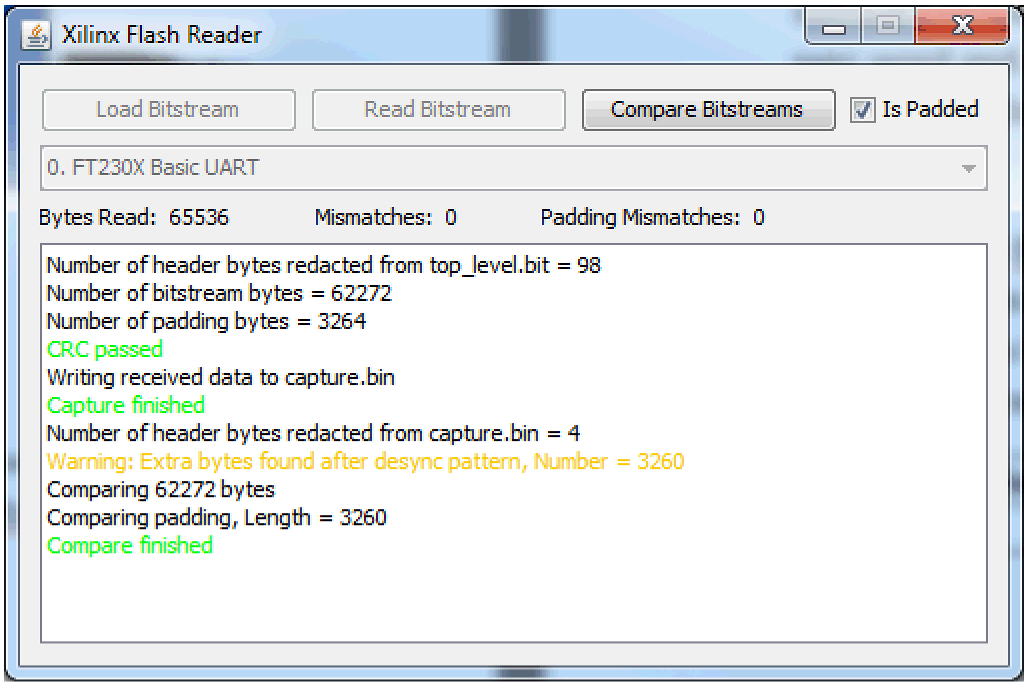
```
Number of header bytes redacted from top_level.bit = 98
Number of bitstream bytes = 62272
Number of padding bytes = 3264
CRC passed
Writing received data to capture.bin
Capture finished
```

Figure 70. GUI After Successful Read

Once capture is complete, as presented in Figure 70, the inspector initiates a comparison. The inspector is then prompted to load a comparison bitstream and should choose the captured bitstream from the previous read step. The software will once again check that the synchronization and desynchronization patterns are present in the captured bitstream. Also, because this file is a captured bitstream and not one generated during application development, no header data should exist, and no redaction occurs.

The software first compares each byte in the loaded bitstream to each byte in the compared bitstream, ignoring the 0xFF padding that may exist at the end of the compare bitstream. Each byte should match exactly, one to one. Any mismatch will be reported to the inspector. Once comparison of the bitstream data is complete, the software then examines the padding if the inspector indicates that padding exists. Every padding byte should be 0xFF, and any differences can once again be noted to the inspector. Any mismatches in either check are displayed and counted.

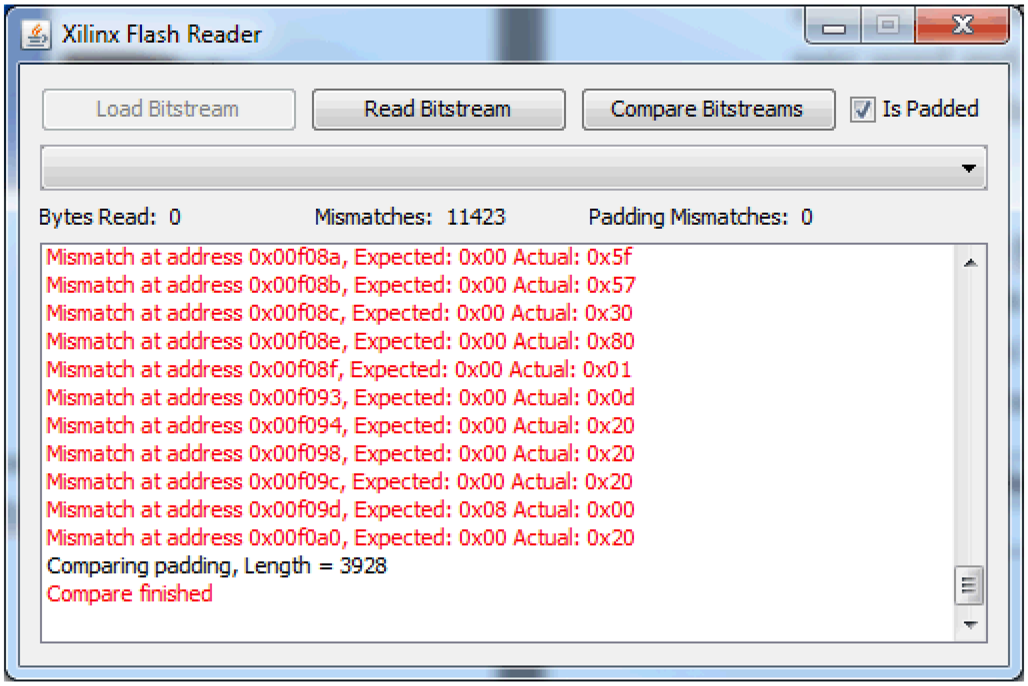
Figure 71 shows a successful bitstream comparison, while Figure 72 shows an unsuccessful.



Bytes Read: 65536 Mismatches: 0 Padding Mismatches: 0

```
Number of header bytes redacted from top_level.bit = 98
Number of bitstream bytes = 62272
Number of padding bytes = 3264
CRC passed
Writing received data to capture.bin
Capture finished
Number of header bytes redacted from capture.bin = 4
Warning: Extra bytes found after desync pattern, Number = 3260
Comparing 62272 bytes
Comparing padding, Length = 3260
Compare finished
```

Figure 71. Successful Bitstream Comparison



Bytes Read: 0 Mismatches: 11423 Padding Mismatches: 0

```
Mismatch at address 0x00f08a, Expected: 0x00 Actual: 0x5f
Mismatch at address 0x00f08b, Expected: 0x00 Actual: 0x57
Mismatch at address 0x00f08c, Expected: 0x00 Actual: 0x30
Mismatch at address 0x00f08e, Expected: 0x00 Actual: 0x80
Mismatch at address 0x00f08f, Expected: 0x00 Actual: 0x01
Mismatch at address 0x00f093, Expected: 0x00 Actual: 0x0d
Mismatch at address 0x00f094, Expected: 0x00 Actual: 0x20
Mismatch at address 0x00f098, Expected: 0x00 Actual: 0x20
Mismatch at address 0x00f09c, Expected: 0x00 Actual: 0x20
Mismatch at address 0x00f09d, Expected: 0x08 Actual: 0x00
Mismatch at address 0x00f0a0, Expected: 0x00 Actual: 0x20
Comparing padding, Length = 3928
Compare finished
```

Figure 72. Unsuccessful Bitstream Comparison

4.10.2 Performance Results

For the baseline design, we have shown that bitstream comparison is a valid technique to ensure that the configuration data loaded onto an SRAM-based FPGA during power-on matches a trusted reference. Further, this technique should scale to larger and more complex designs without serious penalty, as comparison time grows linearly with bitstream size.

One important takeaway is that it is important to have bitstream compression turned on when synthesizing the design. Bitstream compression enables the multiple frame write feature of the FPGA. This feature allows certain frames of configuration data that would be repeated, such as unused logic and routing in the design, to be consolidated in the bitstream itself. This feature allows the FPGA to write identical slices of configuration data to multiple frames at once instead of each frame individually. Depending on how unused the device is, the size of the resulting bitstream can shrink considerably and will vary from design to design. This variation makes it harder for an adversary to alter a design, as the resulting bitstream size will vary from the trusted version.

Evaluation of Original Design

For this test, we compared the trusted bitstream to its unmodified self. 62,370 bytes of configuration data are present in the golden bitstream, and each mapped byte by byte to a corresponding byte in the captured bitstream with zero mismatches. Additionally, 3,162 bytes of padding are needed on the flash memory we chose, and all contained 0xFF.

Evaluation of Modified Designs

We captured and compared all five modified designs, and the results are displayed in Table 7.

Captured Bitstream	Size (bytes)	# of Mismatches in Captured Bitstream
Trusted	62,370	0
Modified Design #1	61,607	11,423
Modified Design #2	62,667	10,863
Modified Design #3	61,027	11,060
Modified Design #4	59,703	11,011
Modified Design #5	59,175	10,729

Table 7. Bitstream Comparison Results

Every modified bitstream was immediately identifiable as altered. The size of each modified bitstream differed from the golden by 1-5% (due to bitstream compression), while the number of mismatches was always over 10,000.

Refutability

At first glance, the technique is not refutable. Any modification to bitstream data will be detected by simple comparison. Further, even tiny changes in the source HDL result in massive changes in the bitstream data. Each modified design had at least 11,000 byte differences versus the trusted design. With clever floorplanning and layout constraints, an adversary could conceivably reduce the number of mismatches by forcing similar logic to go into the same areas, but the number of mismatches will never go to 0.

Nevertheless, this technique relies on physical authentication of both the FPGA and the non-volatile memory hardware. The inspector needs assurance that no malicious configuration data is hidden inside either device in a way that would be inaccessible to our interrogation tools. This hidden data could be loaded surreptitiously after the FPGA has completed a “normal” configuration. Alternatively, an adversary might embed a configuration memory chip within the FPGA itself and simply ignore any configuration data it receives from a valid external source.

4.10.3 Cost Estimation

The equipment for this method exists and would cost less than $200 per unit to construct more. No additional costs are necessary, and the labor involved to run comparisons is trivial.

4.11 Bitstream Comparison (Flash)

This technique is the functional equivalent of bitstream comparison for SRAM FPGAs (discussed in Section 4.10), such as the Xilinx FPGA in our basis system design, but for flash-based FPGAs, such as the Microsemi FPGA in our basis system. However, because flash-based devices are non-volatile and contain the bitstream on the part itself rather than an external memory, the inspector must use a different authentication method.

In order to validate the integrity of the bitstream, the inspector will have to connect a JTAG[17] (or equivalent) controller to the device and read out the contents of the FPGA's internal bitstream memory. Once read out, the bitstream can be compared to a golden, or trusted, copy as before.

For our baseline design employing a Microsemi FPGA, this technique was feasible but not trustable enough to be usable. Microsemi devices do not allow the contents of the bitstream to be extracted from the FPGA and stored locally. For older devices, the JTAG controller sends one row of comparison bitstream data at a time into the device. The device then checks this received data against its own internally programmed data and returns a single bit PASS/FAIL status for each row. For newer devices, the JTAG controller reads one word of bitstream data at a time out of the FPGA and compares it against expected data. In theory, a custom JTAG controller could be designed to mimic this operation but actually store the received data locally, but the procedures to do so are proprietary to Microsemi. In both cases, the only outcome available to the inspector is a green or red light indicating pass or fail. As a result, the inspector has no way to independently verify the authenticity of the FPGA programming and is at the mercy of trusting Microsemi tools and software.

5. Conclusions

There are several reasons to consider the use of FPGAs in treaty verification equipment – ease of development, the ability to exclude unnecessary functions and interfaces, and low cost chief among them. With these advantages comes the challenge of authenticating FPGAs and the systems that they reside in. We have considered building trust in the inspector throughout the development lifecycle of the FPGA system, from application development, to hardware development, through integration and system operation. At each stage of this lifecycle, there are opportunities for the inspector to authenticate the design or the system itself. While the space of potential authentication methods is large (larger even than the 28 methods we propose in Appendix A), we have developed prioritized for studying these methods based on our perceived value and their research cost. Based on these results, the priorities we outlined early on should be reexamined in conjunction with our sponsoring agencies.

In this initial study of FPGA authentication for treaty applications, we were able to investigate (whether by brief survey or applying the method to the basis system as a test case) ten methods for authentication of FPGAs at all three major phases of the development cycle – application development, hardware development, and system operation. The most significant results were gathered in evaluations of FPGA application design verification, both in formal methods to trust that the written HDL meet requirement and in formal equivalence checking to trust that synthesized netlists are functionally equivalent to the trusted HDL, and in verifying that the bitstream that is loaded onto the FPGA is identical to a trusted reference of the bitstream. These are by no means definitive results, especially for the application design verification methods, as they have only been evaluated on a very simple application and only with five "true negative" test cases using intentionally modified designs. We feel that this is the beginning of an investigation that should be continued to further strengthen these authentication methods.

[17] JTAG stands for Joint Test Action Group, an industry association formed for developing standards for design verification of electronics. It is commonly used now as a term to denote the port on processor or board that is used to transfer debug, diagnostic, or programming information in a serial fashion. In this case, the FPGA JTAG port is used to configure the chip and to receive diagnostic information from the chip.

APPENDIX A. FPGA AUTHENTICATION METHODS

A.1. Format

Each potential authentication method is described in a paragraph (intended to be useful to both a technical and a non-technical audience) and in a table that contains common fields. Those fields are described below.

A.1.1. Applicable Technology

Some methods are performed on the built system, and these typically apply to only one type of field programmable gate array (FPGA) that is used in the system. The three types of FPGAs are static random-access memory (***SRAM***), ***Flash***, and one-time programmable (OTP), also known as ***antifuse***. Both SRAM- and Flash-based FPGAs are reconfigurable, whereas the antifuse FPGA is not. The SRAM-based FPGA uses an external configuration memory and does not store the bitstream in non-volatile memory on-chip, whereas the Flash-based FPGA does store the bitstream on-chip. Some methods applied on the built system are not applied to the FPGA, and in those cases they are described within the section. Other methods are performed on design elements. Design methods could be performed on register transfer level (***RTL*** – the hardware description language code), ***netlist***, or ***bitstream***.

A.1.2. Applicable authentication mode

Authentication is either performed on the ***design*** or on the built system. We define five modes of authentication for the built system. Note that "joint analysis" implies the inspector analyzes the equipment in the presence of the host and "private analysis" implies the inspector analyzes the equipment without the presence of the host, in inspector-controlled space.

1. ***Joint analysis before use***
2. ***Joint analysis during use***
3. ***Joint analysis after use*** (assuming the equipment will never be used again)
4. ***Private analysis***, which can either be:
    a. ***By association*** (examining a different but equivalent piece of equipment) or
    b. ***Direct*** (examining the exact piece of equipment that will be or has been used)

A.1.3. Potential value of method

The value of each method is rated on a ***1-5*** scale where "1" is extremely valuable and "5" is not valuable. This subjective rating takes into account the difficulty and consequence of the subversion that would be detected with the method. This parameter was used for a rough idea of priority early in the process.

A.1.4. Research cost of method

The research cost of each method is rated on a ***1-5*** scale where "1" is inexpensive (<$10k) and "5" is very expensive (>$100k). We considered rating each method on the actual cost to perform the method in a treaty context, but determined that we do not have enough information to accurately gauge that cost. Instead, we use the cost to perform the research to assist us in prioritizing the methods (in addition to the potential value of the method). Note that the methods are listed in order of the sum of the potential value and research cost (since "1" is the best in both cases, the lower sums appear first in this document). While these were used to assess priority, in many cases they did not translate directly to priority, as other factors (such as research value or access to specialized tools) were included in the prioritization decisions.

A.1.5. Proposed effort

The team first determined the appropriate course of action for each method in three categories: ***apply*** the method to the FPGA basis system, ***research*** the method in the laboratory to understand the capability better, or ***survey*** the method by conducting a literature review and interviewing subject matter experts. The research cost listed in the previous section is the cost to apply, research, or survey the method, and the cost is naturally dependent on which of those is chosen. For example, a survey is likely to be less expensive to perform than the application of a method. Then the team determined which part of the team (U.S. ***SNL*** or UK ***AWE***) is better suited to perform each of the appropriate actions for each method. For ease of reference, methods assigned to SNL are colored **blue** and methods assigned to AWE are colored **red**.

A.1.6. Priority

To facilitate planning the authentication method evaluations within the constraints of existing resources and time, each lab prioritized the methods assigned to them. There are two separate priority lists—one for SNL and one for AWE. The SNL methods are prioritized in a ranked list starting with "1" as the highest priority. The AWE methods are given a priority of low, medium, or high. Methods with a higher priority are more likely to be evaluated in FY16. Priority is assigned using a potential value, research cost, and other method-specific factors.

A.2. Application Development Authentication Methods

A.2.1. Bitstream authentication

The bitstream is the program data that is loaded onto the FPGA upon configuration. The bitstream sets the hardware assignment and routing and creates the application-specific circuit within the FPGA. It uses a proprietary encoding that varies by FPGA vendor. Bitstream authentication is the process of verifying that the bitstream represents the original RTL, or high-level code, that has been designed to meet the function of the system. A method for verifying the bitstream has a high value, since bitstream generation is the most opaque part of the design process and the bitstream gets loaded directly onto the FPGA.

Applicable technology	Bitstream
Applicable authentication mode	Design
Potential value of method	1
Research cost of method	1
Proposed effort	Survey AWE
Priority	High

A.2.2. Formal methods with Solidify

Solidify uses mathematical verification techniques to confirm or refute specified functional behavior. Any refuted behavior is demonstrated by generating a counterexample showing how such a failure arises.

Applicable technology	RTL and netlist
Applicable authentication mode	Design
Potential value of method	1
Research cost of method	3
Proposed effort	Apply AWE
Priority	High

A.2.3. Constrained random testing with Open Source VHDL Verification Method

Open Source VHDL Verification Method (OSVVM) is an industry standard for functional verification of digital hardware. OSVVM test benches replace traditional test benches and consist of reusable verification components that perform constrained random, coverage-driven verification. Unlike traditional test benches that wiggle wires in a prescribed manner, OSVVM testing applies non-arbitrary (constrained) random stimulus to a design free of the selective bias of a human writer. Given enough time, this style of stimulus will cover an increasing proportion of the entire space of the design and uncover unexpected design behavior.

Applicable technology	RTL
Applicable authentication mode	Design
Potential value of method	2
Research cost of method	2
Proposed effort	Research AWE
Priority	Medium

A.2.4. Formal equivalence check (other than Onespin and Formality)

Formal equivalence checking is a mathematical technique to prove that two representations of a logic circuit exhibit exactly the same functional behavior, i.e., when stimulated with any valid sequence of inputs, both designs produce exactly the same outputs. Formal equivalence checking will ensure that no unexpected logic was inserted into the design or intended logic was removed from it. This survey will investigate equivalence checking approaches other than Onespin and Formality.

Applicable technology	RTL and netlist
Applicable authentication mode	Design
Potential value of method	3
Research cost of method	1
Proposed effort	Survey AWE
Priority	Medium

A.2.5. Static analysis

Static analysis considers the semantics of the RTL (rather than its execution) to determine whether it fulfills certain properties of interest. State of the art technologies (such as SMT solvers) could be applicable to the static analysis of VHDL.

Applicable technology	RTL
Applicable authentication mode	Design
Potential value of method	2
Research cost of method	3
Proposed effort	Research AWE
Priority	Low

A.2.6. Formal methods with Onespin

Formal verification is a mathematical technique that proves that a given logic design meets a set of precisely expressed functional behaviors. Onespin[18] is a commercial tool that can exhaustively verify that a design does everything it is supposed to do and does not do anything that it is not specified to do if the assertions are written properly to support this.

Applicable technology	RTL and netlist
Applicable authentication mode	Design
Potential value of method	1
Research cost of method	3
Proposed effort	Apply SNL
Priority	1

A.2.7. Formal equivalence check with Onespin

Formal equivalence checking is a mathematical technique to prove that two representations of a logic circuit exhibit exactly the same functional behavior, i.e., when stimulated with any valid sequence of inputs, both designs produce exactly the same outputs. Because FPGA designs are a multi-step process whereby the underlying representation of the circuit changes from step to step, formal equivalence checking will ensure that no unexpected logic was inserted into the design or intended logic was removed from it. Onespin is an FPGA-specific tool for formal equivalence checking.

Applicable technology	RTL and netlist
Applicable authentication mode	Design
Potential value of method	1
Research cost of method	3
Proposed effort	Apply SNL
Priority	2

A.2.8. Formal equivalence check using Formality

Formal equivalence checking is a mathematical technique to prove that two representations of a logic circuit exhibit exactly the same functional behavior, i.e., when stimulated with any valid sequence of inputs, both designs produce exactly the same outputs. Because FPGA designs are a multi-step process whereby the underlying representation of the circuit changes from step to step, formal equivalence checking will ensure that no unexpected logic was inserted into the design or intended logic was removed from it. This method uses a commercial tool call Formality[19], which is an ASIC-specific tool.

[18] URL: https://www.onespin.com
[19] URL: http://www.synopsys.com/Tools/Verification/FormalEquivalence/Pages/Formality.aspx

Applicable technology	RTL and netlist
Applicable authentication mode	Design
Potential value of method	1
Research cost of method	4
Proposed effort	Apply SNL
Priority	3

A.2.9. Constrained random testing with Universal Verification Methodology (UVM)

UVM[20] is an industry standard for functional verification of digital hardware. UVM test benches replace traditional test benches and consist of reusable verification components that perform constrained random, coverage-driven verification. Unlike traditional test benches that wiggle wires in a prescribed manner, UVM testing applies non-arbitrary (constrained) random stimulus to a design free of the selective bias of a human writer. Given enough time, this style of stimulus will cover an increasing proportion of the entire space of the design and uncover unexpected design behavior.

Applicable technology	RTL
Applicable authentication mode	Design
Potential value of method	2
Research cost of method	1
Proposed effort	Survey SNL
Priority	6

A.2.10. Simulation test bench functional testing

A functional test bench is a tool for simulating the behavior of a design to determine the correctness of its logic circuit. Test benches are user generated designs that apply stimulus to the application and collect output responses. These output responses are then compared to expected values and verified. By writing multiple, comprehensive test benches, the inspector may be able to exercise all possible lines of code and branch paths within the application to help ensure that no unexpected behavior exists in the source RTL or any synthesized netlists.

[20] URL: http://accellera.org/downloads/standards/uvm/

Applicable technology	RTL and netlist
Applicable authentication mode	Design
Potential value of method	1
Research cost of method	2
Proposed effort	Apply SNL
Priority	7

A.2.11. Code coverage simulation test bench

Code coverage is a set of metrics to evaluate the thoroughness of a test bench. Coverage metrics include line coverage (has every line in the source code been executed?), branch coverage (has each branch in a control statement been executed?), and condition coverage (has each Boolean sub-expression been evaluated as both true and false?). This technique will result in the creation of test benches that provide 100% coverage of the three different metrics.

Applicable technology	RTL
Applicable authentication mode	Design
Potential value of method	2
Research cost of method	2
Proposed effort	Apply SNL
Priority	9

A.2.12. Static analysis of source code

Static analysis is an automated analysis of source code that is executed without actually running the code. Static analysis techniques, such as linting, algorithmically analyze source code and flag potential suspicious language usage. Such language constructs could potentially be used to hide subversive logic in an application.

Applicable technology	RTL
Applicable authentication mode	Design
Potential value of method	3
Research cost of method	2
Proposed effort	Research SNL
Priority	12

A.2.13. Manual code inspection

This technique requires a subject matter expert to manually analyze the source code for proper functionality as well as the absence of subterfuge and hidden functionality. Code should be commented thoroughly and written to a standard to facilitate this style of inspection.

Applicable technology	RTL
Applicable authentication mode	Design
Potential value of method	5
Research cost of method	3
Proposed effort	Apply SNL
Priority	14

A.3. Hardware Development Authentication Methods

A.3.1. Depackaging and imaging (Flash)

Depackaging allows analysis of the silicon in an integrated circuit to be examined. It can show if the die has been replaced with an entirely different die, the die has been tampered using focused ion beam (FIB) modification, or an extra die has been added to the package. With removal of successive layers of the die we can see if a new silicon die has been created with modified masks.

Applicable technology	Flash
Applicable authentication mode	Private analysis
Potential value of method	1
Research cost of method	3
Proposed effort	Research AWE
Priority	High

A.3.2. Depackaging and imaging (SRAM)

Depackaging allows analysis of the silicon in an integrated circuit to be examined. It can show if the die has been replaced with an entirely different die, the die has been tampered using FIB modification, or an extra die has been added to the package. With removal of successive layers of the die we can see if a new silicon die has been created with modified masks.

Applicable technology	SRAM (FPGA only, not external PROM)
Applicable authentication mode	Private analysis
Potential value of method	2
Research cost of method	2
Proposed effort	Research AWE
Priority	High

A.3.3. Visual and X-ray examination of printed circuit board

Printed circuit boards (PCBs) will be examined with populated components. Visually examining the PCB assembly gives a low level of confidence that the correct parts were used and that there are correct routes between the parts. Adding additional imaging with X-ray examination gives an increased (but still low) confidence of the circuit design. X-ray imaging will expose hidden components and traces. In addition, automated tools exist for checking PCB build quality in a production line and we will explore if they are applicable to our problem.

Applicable technology	PCB
Applicable authentication mode	Joint analysis before use, joint analysis after use, private analysis
Potential value of method	1
Research cost of method	4
Proposed effort	Apply AWE
Priority	High

A.3.4. Destructive printed circuit board examination

X-ray analysis may not have the resolution to examine fine details of the PCB. Vertical cuts at a variety of location along the PCB or delaminating successive layers followed by close examination with a microscope may address this issue.

Applicable technology	PCB
Applicable authentication mode	Private analysis
Potential value of method	4
Research cost of method	1
Proposed effort	Survey AWE
Priority	Medium

A.3.5. Depackaging and imaging (OTP/antifuse)

Depackaging allows analysis of the silicon in an integrated circuit to be examined. It can show if the die has been replaced with an entirely different die, the die has been tampered using FIB modification, or an extra die has been added to the package. With removal of successive layers of the die we can see if a new silicon die has been created with modified masks.

Applicable technology	Antifuse
Applicable authentication mode	Private analysis
Potential value of method	3
Research cost of method	3
Proposed effort	Research AWE
Priority	Low

A.3.6. Depackaging and imaging (external PROM)

Depackaging allows analysis of the silicon in an integrated circuit to be examined. It can show if the die has been replaced with an entirely different die, the die has been tampered using FIB modification, or an extra die has been added to the package. With removal of successive layers of the die we can see if a new silicon die has been created with modified masks.

Applicable technology	SRAM (external PROM only)
Applicable authentication mode	Private analysis
Potential value of method	1
Research cost of method	3
Proposed effort	Research AWE
Priority	Low

A.3.7. Manual hardware design verification

The hardware design could be manually verified by ensuring that the schematics, bills of material, and Gerber files (PCB layer files) do not contain hidden components or pathways that could introduce unexpected functionality.

Applicable technology	Hardware designs (BoM, schematic, Gerber)
Applicable authentication mode	Design
Potential value of method	3
Research cost of method	3
Proposed effort	Apply AWE
Priority	Low

A.3.8. Scan chain verification (SRAM)

Scan chain verification is a type of testing designed to exercise all possible logic elements that exist within a piece of digital hardware. This technique is used to verify the bare metal hardware of the FPGA, not the application design. All synchronous elements in the FPGA are connected together sequentially. The input to this register chain is stimulated using an automatic test pattern generator and the response verified. Any faulty logic elements or unspecified hardware should be detected during this testing. Because SRAM FPGAs have removable configuration memory, the inspector would replace the host configuration memory with his own memory chip containing the scan chain design. For this reason, it is probably not acceptable to be used for direct authentication before use.

Applicable technology	SRAM
Applicable authentication mode	Joint analysis before use, joint analysis after use
Potential value of method	2
Research cost of method	3
Proposed effort	Apply SNL
Priority	11

A.3.9. Scan chain verification (Flash)

This technique is identical to scan chain verification for SRAM devices, except that the Flash FPGA must be reprogrammed since it does not have an external configuration memory.

Applicable technology	Flash
Applicable authentication mode	Joint analysis after use
Potential value of method	3
Research cost of method	3
Proposed effort	Apply SNL
Priority	13

A.4. System Operation Authentication Methods

A.4.1. Power analysis

Power analysis has been used in many security-related areas to understand the functionality of a system. It is a non-invasive technique that may uncover unexpected functionality by analyzing the power drawn by the system during operation or while the system is not operating but is powered on.

Applicable technology	All FPGA types (built system)
Applicable authentication mode	Joint analysis before use, private analysis
Potential value of method	4
Research cost of method	3
Proposed effort	Research AWE
Priority	Low

A.4.2. Bitstream monitoring during configuration (SRAM)

In addition to bitstream comparison (SRAM), the bitstream can also be monitored in-circuit as the FPGA is configuring itself. This technique would require an electrically isolated interface to the configuration lines on the printed circuit board to mitigate host concerns that the inspector might try to alter the FPGA configuration or inject data into the device to reveal sensitive information.

Applicable technology	SRAM
Applicable authentication mode	Joint analysis before use, joint analysis after use, private analysis
Potential value of method	1
Research cost of method	2
Proposed effort	Apply SNL
Priority	4

A.4.3. Bitstream comparison (SRAM)

Bitstreams for SRAM-based FPGAs are stored in external non-volatile memory. When the FPGA is powered on, it reads the contents of this memory and configures itself accordingly. If this memory is socketed (meaning that it is inserted into a socket in such a way as to be easily removed by hand, rather than being soldered to the board), the inspector can remove the memory post-inspection and perform analysis on it. One such analysis is a readout of the contents of the device and comparison to a golden, or trusted, copy.

Applicable technology	SRAM
Applicable authentication mode	Joint analysis after use, private analysis
Potential value of method	1
Research cost of method	2
Proposed effort	Apply SNL
Priority	5

A.4.4. Bitstream comparison (Flash)

Unlike SRAM FPGAs which use an external memory to hold the bitstream, Flash-based devices contain the bitstream on the part itself. In order to validate the integrity of the bitstream, the inspector will have to connect a JTAG[21] cable to the device and read out the contents of the FPGA's internal bitstream memory. Once read out, the bitstream can be compared to a golden, or trusted, copy as before.

Applicable technology	Flash
Applicable authentication mode	Joint analysis after use, private analysis
Potential value of method	2
Research cost of method	1
Proposed effort	Apply SNL
Priority	5

A.4.5. Bitstream real time monitor with built-in CRC or hash

Because SRAM FPGAs configure in a synchronous serial manner, real-time monitoring of the configuration data could be accomplished via built-in hashing or cyclic redundancy check (CRC) hardware, which would include the hardware to calculate the hash or CRC and a display to show the result. This approach is less intrusive than bitstream monitoring during verification as no inspector equipment will touch the host hardware. The result of the hash or CRC will be visually inspectable after configuration completes.

Applicable technology	SRAM
Applicable authentication mode	Joint analysis during use, private analysis
Potential value of method	2
Research cost of method	2
Proposed effort	Research SNL
Priority	8

[21] JTAG stands for Joint Test Action Group, which implements standards for on-chip instrumentation. The standard itself has been adopted as the common name for the programming port for microcontrollers and other programmable devices, like FPGAs.

A.4.6. Functional test of hardware with real or simulated inputs

While simulations are great for controlled testing, they are slow to execute. As a result, comprehensively testing an application can take unreasonably and sometimes impossibly long amounts of time (that is, the simulation tool will stop trying to complete the task). A solution to this problem is to test the design in hardware with real inputs. These inputs could come from a controlled stimulator or actual hardware, such as a radiation detector.

Applicable technology	All FPGA types (built system)
Applicable authentication mode	Joint analysis before use, joint analysis after use, private analysis
Potential value of method	1
Research cost of method	3
Proposed effort	Apply SNL
Priority	10

APPENDIX B. FPGA BASIS SYSTEM LOGICAL DESIGN

The logical design of the portal monitor application is shown in Figure 73. Each of the boxes in the diagram is described below. The detailed physical and logical design of the FPGA basis system, including schematics for two boards (one using a Xilinx FPGA and one using a Microsemi FPGA), PCB layouts for the two boards, bills of materials for the two boards, and the VHDL RTL are included in an addendum.

Each of the blocks in Figure 73 represents one .VHD file in the VHDL design. In each of the descriptions, the names of the generics, inputs, and outputs are the same as in the VHDL files.

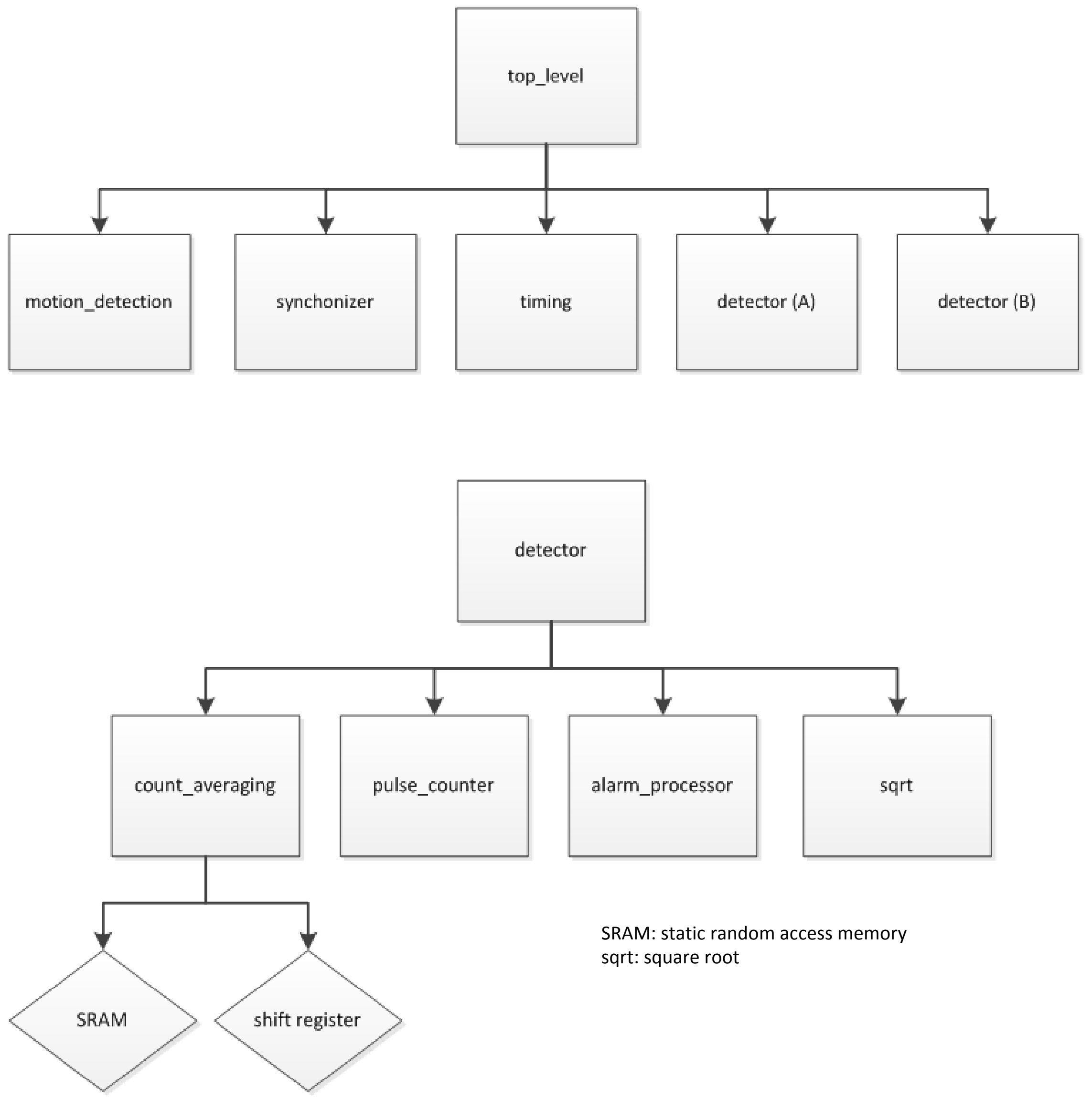


Figure 73. Logical Design of the Portal Monitor Application

B.1. Top Level Block

Inputs

- **Clock** – 150 KHz external clock input
  - Sourced externally
- **Reset** – System reset signal from voltage supervisor. Active-low. Resets all device registers
  - Sourced externally
- **Push_button_reset** – Manual reset push button. Active-high. Used to reset the detector alarms
  - Sourced externally
- **Pulse_A**, **Pulse_B** – Digital pulse inputs from detectors
  - Sourced externally

Outputs

- **AtoB_alarm**, **BtoA_alarm** – Active-high direction of material movement alarms
  - Sourced by motion_detected.vhd
- **High_background_alarm** – Active-high alarm indicating background level has exceeded predefined level
  - Sourced by alarms.vhd
- **Low_background_alarm** - Active-high alarm indicating background level has dropped below predefined level
  - Sourced by alarms.vhd
- **Ready_LED** – Indicates detector readiness. Glows solidly on power-up and when count averaging circuit has not had time to fill up. Blinks otherwise indicating detector is functional.
  - Sourced by timing.vhd

Behavior

This file is the top level of the design. It is a structural file that is used to connect together the synchronizer, timer, detector modules, and motion detector. The only logic included is the ANDing of the A/B detector ready signals, and the ORing of the A/B background high/low alarm signals.

B.2. Synchronizer Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal

Inputs

- **Clock –** 150 kHz system clock
  - Sourced externally
- **Reset –** System reset
  - Sourced externally
- **Push_button_reset –** Manual reset button
  - Sourced externally
- **Pulse_A, Pulse_B –** Digital pulse inputs from detectors
  - Sourced externally

Outputs

- **Reset_sync** – Synchronized version of the Reset input
    - Sourced by this module
- **Push_button_sync** – Synchronized version of the Push_button_reset input
    - Sourced by this module
- **Pulse_A_sync, Pulse_B_sync** – Synchronized versions of the Pulsa_A, Pulse_B inputs
    - Sourced by this module

Behavior

This file synchronizes all external input to the *Clock* signal using a double registering scheme to prevent metastability. The signal values are asynchronously reset on assertion of the *Reset* input. On release of *Reset*, the true value of the signal passes through a set of two shift registers and is then used by the rest of the design.

B.3. Timer Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal

Inputs

- **Clock** – 150 kHz system clock
    - Sourced externally
- **Reset** – System reset
    - Sourced externally
- **Detector_ready** – ANDed version of the A/B detector ready signals
    - Sourced by top_level.vhd

Outputs

- **Clear_pulse_counter** – Active-high output the clears the pulse counts on a regular interval
    - Sourced by this module
- **Ready_LED** – Indicates detector readiness. Glows solidly on power-up and when count averaging circuit has not had time to fill up. Blinks otherwise indicating detector is functional.
    - Sourced by this module

Behavior

This file has two functions. First, it provides the clear signal for the pulse counting modules. The time interval is defined as a constant (COUNTING_INTERVAL) in the file Constants.vhd. For example, a value of 14999 for COUNTING_INTERVAL will result in a time interval of 100 milliseconds. The counting modules will count the number of input pulses received for the duration of this interval. We chose a 150 kHz clock to limit the maximum value of this counter in the hopes that it would make formal verification easier by limiting the number of bits in the counter. The second function of this file is to provide the detector ready output. This output glows steadily when the device has been reset via the system reset signal. It also glows steadily when the device is first running and accumulating an average background level. Once the background has been established, this output blinks at the rate defined by COUNTING_INTERVAL. The blinking functionality lets the inspector know that the device is running and the clock is functional.

B.4. Motion Detection Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal

Inputs

- **Clock –** 150 kHz system clock
  - Sourced externally
- **Reset –** System reset
  - Sourced externally
- **Push_button_reset –** Manual reset button
  - Sourced externally
- **Material_alarm_A, Material_alarm_B –** pulse inputs from detectors
  - Sourced by detector.vhd

Outputs

- **AtoB_alarm**, **BtoA_alarm** – Active-high direction of material movement alarms
  - Sourced by this module

Behavior

This module contains the logic for determining if material moved from A to B or from B to A. On reset, both alarms are asserted. This condition is an error condition to show that power up has occurred, and it can only be removed via assertion of the pushbutton reset. Once the alarms are cleared, the module monitors the material alarms from each detector. It waits to see which is asserted first and then waits for the opposite alarm to occur. If A occurs first, the direction is A to B, and if B occurs first, the direction is B to A. An indefinite amount of time can take place between the two alarms and only one alarm can trigger (unless power on has occurred). Also, once either direction alarm has asserted, it can only be cleared via pushbutton reset.

B.5. Detector Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal
- **COUNT_WIDTH** – Defines the number of bits of the pulse counter
- **BUFFER_WIDTH** – Defines the number of elements in the pulse count averaging logic (2^BUFFER_WIDTH)

Inputs

- **Clock –** 150 kHz system clock
  - Sourced externally
- **Reset –** System reset
  - Sourced externally
- **Push_button_reset –** Manual reset button
  - Sourced externally
- **Pulse** – Digital pulse input from detector

- Sourced externally
- **Clear_pulse_counter** – Active-high output the clears the pulse counts on a regular interval
  - Sourced by timing.vhd

Outputs

- **Detector_ready** – Active-high signal that count averaging logic has filled its buffer and thus obtained a valid average
  - Sourced by count_averaging.vhd
- **Material_alarm** – Active-high indication that the current count capture value exceeds a certain threshold (i.e. material is present near the detector)
  - Sourced by alarm_processor.vhd
- **High_background_alarm** – Active-high alarm indicating background level has exceeded predefined level
  - Sourced by alarm_ processor.vhd
- **Low_background_alarm** - Active-high alarm indicating background level has dropped below predefined level
  - Sourced by alarm_ processor.vhd

Behavior

This module is a structural design that represents a single detector. Two detector modules are instantiated in the design to represent the A & B detectors. The only logic included is a rising and falling edge detector for the material alarm signal. A rising edge pauses the averaging circuit, while a falling edge unpauses.

B.6. Count Averaging Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal
- **COUNT_WIDTH** – Defines the number of bits of the pulse counter
- **BUFFER_WIDTH** – Defines the number of elements in the pulse count averaging logic (2^BUFFER_WIDTH)

Inputs

- **Clock –** 150 kHz system clock
  - Sourced externally
- **Reset –** System reset
  - Sourced externally
- **Load –** Active high signal to load a new count value in
  - Sourced by timing.vhd (Clear_pulse_counter)
- **Pulse** – Digital pulse input from detector
  - Sourced externally
- **Pulse_count** – A count of the number of pulses per time interval
  - Sourced by pulse_counter.vhd

Outputs

- **Valid** – Active-high signal that count averaging logic has filled its buffer and thus obtained a valid average
    - Sourced by this module
- **Average –** The average value of the pulse count over an interval defined by BUFFER_WIDTH.
    - Sourced by this module

Behavior

This module averages the pulse count for a specified interval. Initially the buffer is empty and valid is negated. Each time *load* is asserted (assuming *pause* is inactive), the module inserts a new count value into the buffer. Once the buffer has filled, *Valid* is asserted. At this point, when a new count values is inserted, the oldest count value in the buffer is removed. If *pause* is ever asserted, the buffer (and the resulting average value) will be frozen until pause is removed.

This module is implemented using two different architectures. The first architecture uses a shift register to implement the buffer, while the second architecture uses an SRAM.

B.7. Pulse Counter Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal
- **WIDTH** – Defines the number of bits of the pulse counter

Inputs

- **Clock –** 150 kHz system clock
    - Sourced externally
- **Reset –** System reset
    - Sourced externally
- **Pulse** – Digital pulse input from detector
    - Sourced externally
- **Clear** – Active-high output the clears the pulse counts on a regular interval
    - Sourced by timing.vhd

Outputs

- **Count** - A count of the number of pulses per time interval
    - Sourced by this module

Behavior

This module is a simple pulse counter. If *clear* is asserted, the pulse count is reset to 0. Otherwise, if a rising edge occurs on *Pulse*, the count is incremented by one. The module also includes logic to ensure that the count cannot overflow.

B.8. Alarm Processor Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal
- **COUNT_WIDTH** – Defines the number of bits of the pulse counter

Inputs

- **Clock –** 150 kHz system clock
    - Sourced externally
- **Reset –** System reset
    - Sourced externally
- **Push_button_reset –** Manual reset button
    - Sourced externally
- **New_count** – Active-high signal asserted when a new count value is loaded in
    - Sourced by timing.vhd (Clear_pulse_counter)
- **Average_valid** – Active-high signal that count averaging logic has filled its buffer and thus obtained a valid average
    - Sourced by count_averaging.vhd
- **Sigma** – The square root of the average count value
    - Source by sqrt.vhd
- **Count** – A count of the number of pulses per time interval
    - Sourced by pulse_counter.vhd
- **Average** – The average value of the pulse count over an interval
    - Sourced by count_averaging.vhd

Outputs

- **Material_alarm** – Active-high indication that the current count capture value exceeds a certain threshold (i.e. material is present near the detector)
    - Sourced by this module
- **High_background_alarm** – Active-high alarm indicating background level has exceeded predefined level
    - Sourced by this module
- **Low_background_alarm** - Active-high alarm indicating background level has dropped below predefined level
    - Sourced by this module

Behavior

This module asserts the various alarms in the design. On system reset, all alarm conditions are asserted. This condition is to prevent someone from powering the device off and on to clear alarms. All alarms can only be cleared via the push button reset. Otherwise, the alarms are asserted as follows: *Average_valid* must be asserted for any alarm to go off. The material alarms asserts when the current count exceeds the average count + 4*sigma. The high/low background alarms assert when the average count exceeds/falls below thresholds defined in Constants.vhd

B.9. Square Root Block

Generics

- **RESET_LEVEL** – Defines the activation level of the system reset signal
- **WIDTH** – Defines the number of bits of the pulse counter

Inputs

- **Clock –** 150 kHz system clock
    - Sourced externally
- **Reset –** System reset
    - Sourced externally
- **Start** – Signal to start the update the calculated square root
    - Sourced by timing.vhd (Clear_pulse_counter)
- **a** – The number to be square rooted
    - Sourced by pulse_counter.vhd (Count)

Outputs

- **Done** – Active-high signal that asserts when a square root operation completes
    - Sourced by this module
- **root** – The square root of a
    - Sourced by this module

Behavior

This module calculates the square root of a number on the rising edge of Start using the following algorithm:

```
unsigned long sqrt(unsigned long a) {
  unsigned long square = 1;
  unsigned long delta = 3;
  while(square <= a) {
    square = square + delta;
    delta = delta + 2;
  }
  return (delta/2 - 1);
}
```

Once the algorithm completes, *Done* is asserted. It then waits for *Start* to be negated before a new square root can be calculated.

APPENDIX C. FPGA BASIS SYSTEM PCB BILLS OF MATERIALS

Table 8. Bill of Materials for the Microsemi Design

Part Number	Description	Designator	Quantity
C1005X7R1C104K050BC	CAP CER 0.1UF 16V X7R 10%	C1, C2, C3, C4, C5, C6, C7, C8, C9, C14, C15, C17	12
CGJ3E2C0G1H103J	CAP CER 0.01UF 50V C0G 5% 0603	C12	1
C1608X5R1E334K080AC	CAP CER 0.33UF 25V X5R 10% 0603	C13	1
CL10C103JA8NNNC	CAP CER 10NF 25V C0G 5% 0603	C16	1
HSMG-C190	Surface Mount Chip LED, Green	D1, D2	2
HSMC-C190	High Performance Chip LED, Red	D3, D4	2
HSMD-C190	Surface Mount Chip LED, Orange	D5	1
Fiducial	Fiduciary Mark	FM1, FM2, FM3	3
5-1814832-1	SMA Female	J1, J2	2
5-103635-2	Flat Flex Cable, 3 pin, R/A	P1	1
3220-10-0300-00	Header, Female, 10 pos, 0.05"	P2	1
RC1608F102CS	RES 1KOHM 1% 0.1W	R1, R2	2
ERJ-3EKF2100V	RES 210OHM 1% 0.1W	R3, R4, R5, R6, R7	5
RT0603BRD0713K3L	RES 13.3KOHM 0.1% 0.1W	R9	1
ADTSM62RVTR	SPST Switch	SW1	1
A3PN125-VQG100	ProASIC3 nano Flash FPGA, 71 User IOs, 125K System Gates, Standard Speed, 100-Pin VQFP, Pb-Free, Commerical Grade	U1	1
TPS73115	Regulator, 1.5V	U2	1
LTC6900	Oscillator, Silicon 1kHz to 20MHz	U3	1
TPS3306-15	Supervisor, 1.5V, 3.3V	U4	1
MAX6816	Switch Debouncer	U5	1

Table 9. Bill of Materials for the Xilinx Design

Part Number	Description	Designator	Quantity
C1005X7R1C104K050BC	CAP CER 0.1UF 16V X7R 10%	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13, C17, C18, C19, C20	17
CL10B105KO8NNNC	CAP CER 1.0UF 16V X7R 10%	C14, C15, C16, C21	4
HSMG-C190	Surface Mount Chip LED, Green	D1, D2	2
HSMC-C190	High Performance Chip LED, Red	D3, D4	2
HSMD-C190	Surface Mount Chip LED, Orange	D5, D6	2
Fiducial	Fiduciary Mark	FM1, FM2, FM3	3
87832-1420	14 Pin JTAG Connector	J1	1
5-1814832-1	SMA Female	J2, J3	2
5-103635-6	Flat Flex Cable, 6 pin, R/A	P1	1
5-103635-2	Flat Flex Cable, 3 pin, R/A	P2	1
BSS138-7-F	NFET 50V 0.2A	Q1	1
RC1608J472CS	RES 4.7KOHM 5% 0.1W	R1, R2, R3	3
RC1608J100CS	RES 10OHM 5% 0.1W	R4, R5	2
ERJ-3EKF2100V	RES 210OHM 1% 0.1W	R6, R7, R8, R9, R13, R14	6
RR0816P-3322-D-51C	RES 33.2KOHM 0.5% 0.063W	R10, R15	2
RT0603BRD0713K3L	RES 13.3KOHM 0.1% 0.1W	R11, R12	2
ADTSM62RVTR	SPST Switch	SW1	1
XC3S100E-4VQG100C	Spartan-3E 1.2V FPGA, 66 User I/Os, 100-Pin VQFP, Standard Performance, Commercial Grade, Pb-Free	U1	1
M25P05-A	Memory, Flash, 512K, SPI	U2	1
ISO7640FMDWR	150 Mbps 6 kVpk Low Power Quad Channels, Digital Isolator, 2.7 V / 3.3 V / 5 V, -40 to +125 degC, 16-pin SOIC (DW), Green (RoHS & no Sb/Br)	U3	1
TLV70212DBVR	Single Output LDO, 300 mA, Fixed 1.2 V Output, 2 to 5.5 V Input, with Low IQ, 5-pin SOT-23 (DBV), -40 to 125 degC, Green (RoHS & no Sb/Br)	U4	1
STM6720	Supervisor, 1.2V, 3.3V, Adj	U5	1
LTC6900	Oscillator, Silicon 1kHz to 20MHz	U6	1
MAX6816	Switch Debouncer	U7	1
TLV70225DBVR	Single Output LDO, 300 mA, Fixed 2.5 V Output, 2 to 5.5 V Input, with Low IQ, 5-pin SOT-23 (DBV), -40 to 125 degC, Green (RoHS & no Sb/Br)	U8	1

APPENDIX D. SPECIFICATIONS FOR FORMAL VERIFICATION USING ONESPIN

D.1. Specifications for Formal Verification

The original specifications provided for the modules are given below. The specifications are not complete for all modules. For modules without full specifications, verification was performed against a reasonable expectation of function for the module.

D.1.1. Timer Specification

1. Assert that Clear_pulse_counter goes high for one cycle every COUNTING_INTERVAL cycles.
2. If Detector_ready = 1, assert that Ready_LED_tl toggles every COUNTING_INTERVAL cycles.
3. If Detector_ready = 0, assert that Ready_LED_tl = 1.

D.1.2. Motion Detection Specification

1. Assert that if a Push_button_reset_tl rising edge has occurred in the past and is not currently active, if Material_alarm_A is high followed by Material_alarm_B going high (or simultaneously), AtoB_alarm_tl goes high. Also assert that BtoA_alarm_tl cannot also go high until a Push_button_reset_tl falling edge has occurred or Reset_tl = RESET_LEVEL.
2. Assert that if a Push_button_reset_tl rising edge has occurred in the past, if Material_alarm_B is high followed by Material_alarm_A going high, BtoA_alarm_tl goes high. Also assert that AtoB_alarm_tl cannot also go high until a Push_button_reset_tl falling edge has occurred or Reset = RESET_LEVEL.

D.1.3. Detector Specification

1. Assert that pause_averager_r =1 one cycle after rising edge of material_alarm_int.
2. Assert that pause_averager_r = 0 one cycle after falling edge of material_alarm_int.

D.1.4. Pulse Counter Specification

1. Assert that if Clear_pulse_counter (from Timer) = 1, Count = 0 after one cycle.
2. Assert that if a rising edge of Pulse (A or B from the top level) happens, Count increments by one unless Count has saturated.

D.1.5. Alarm Processor Specification

1. Assert that if Push_button_reset_tl = 0, High_background_alarm_tl goes high if Average >= HIGH_ALARM_LEVEL.
2. Assert that if Push_button_reset_tl = 0, Low_background_alarm_tl goes high if Average <= LOW_ALARM_LEVEL.
3. Assert that if Push_button_reset_tl = 0, Material_alarm goes high if Average_valid = 1 and New_count = 1 and Count > (4*Sigma + Average).

4. Assert that High/Low_background_alarm_tl and Material_alarm only go low if Push_button_reset_tl = 0.

D.1.6. Count Averaging Specification

Count averaging did not have an original specification due to the complexity of the module.

D.1.7. Sqrt Specification

A specification for the module was not given originally, but pseudo-code for the algorithm behind the square root was outlined as a "C" program. Other properties of this module were based on verification expectations.

```
unsigned long sqrt(unsigned long a) {
   unsigned long square = 1;
   unsigned long delta = 3;
   while (square <= a) {
      square = square + delta;
      delta = delta + 2;
   }
   return (delta/2 - 1);
}
```

The designer indicated that the most correct way to check the square root value was to ensure that the output of the square root module, root, was such that:

$$root^2 \leq input \leq (root + 1)^2$$

D.2. Formal Properties

For each module of the design, we wrote formal SystemVerilog Assertions (SVA) to describe the functional behavior. This chapter provides the properties that were checked for each module, along with their formal encoding.

D.2.1. Timing Properties

The signals used in the verification of the timing module are given in Table 10. To prove the properties, we used the inputs and outputs of this module along with one internal signal. The properties given here mirror the specifications given in section D.1.1.

Table 10. Signals used in Verification of the Timing Module

Signal	Type
Detector_ready	Input
Clear_pulse_counter	Output
Ready_LED	Output
count_r	Internal

D.2.1.1. LED Output Before Detector is Ready

If detector is not ready then LED is steady on.

```
property P1a_time_steady_LED_if_not_ready;
   !timing.Detector_ready |-> timing.Ready_LED;
endproperty
```

Specified as written, this property does not pass. In the design, the LED is ON on the next cycle.

```
property P1b_time_steady_LED_if_not_ready;
   !timing.Detector_ready |=> timing.Ready_LED;
endproperty
```

D.2.1.2. Pulsing of the Clear_pulse_counter

This property shows that the clear_pulse_counter pulses every COUNTING_INTERVAL cycles.

The simplest way to describe this would be to show that: "If Clear_pulse_counter asserts, it should low for the next COUNTING_INTERVAL cycles before reasserting". But since COUNTING_INTERVAL is large, verification (with existing tools) is difficult.

```
property P2a_time_clear_pulse_counter_every_N;
   timing.Clear_pulse_counter // If Clear_pulse_counter
      |=> // then on the next cycle
      (!timing.Clear_pulse_counter [*14999] ) // It is low for 14999 cycles
      ##1 timing.Clear_pulse_counter; // And then high on the following cycle
endproperty
```

The tool cannot conclusively verify the property given here. So instead we use the alternate approach below.

We can split this problem into the following two steps which are easy to verify:

1. Increasing count: At least one clock cycle after reset, whenever the clear_pulse_counter output is 0, the count must be increasing.

```
property P2b_time_clear_count_increments;
   1 ##1 // At least one clock cycle after reset
   !timing.Clear_pulse_counter // and clear_pulse_counter output is 0,
      |->
      (timing.count_r == $past(timing.count_r)+1 ); //the count must be increasing.
endproperty
```

2. Clear count: Whenever the timer count is 14999, on the next cycle the count shall be 0 and the clear_pulse_counter output shall be high.

```
property P2c_time_clear_count_when_full_and_pulse_output;
   (timing.count_r == 14999) // When the count reaches 14999
      |=> // On the next cycle
      (timing.count_r == 0) && // count shall be 0 AND
      timing.Clear_pulse_counter; // Clear_pulse_counter shall be high
endproperty
```

D.2.1.3. LED Pulsing

This property shows that the LED driver toggles every COUNTING_INTERVAL cycles if the detector is ready. We write it formally as "If Detector is ready and Clear_pulse_counter is high, then LED toggles." A direct interpretation of this statement fails in the original design.

```
property P3a_time_toggle_LED_should_not_pass;
   timing.Detector_ready &&
   timing.Clear_pulse_counter
      |=>
      (timing.Ready_LED == !$past(timing.Ready_LED)); // toggles
endproperty
```

It turned out that the property written above wasn't sufficient. The detector also has to be ready on the previous clock cycle for the LED to toggle.

```
property P3b_time_toggle_LED;
   timing.Detector_ready // Detector has to be ready
   ##1 // followed by
   timing.Detector_ready && // detector ready and
   timing.Clear_pulse_counter // Clear_pulse_counter
      |->
      (timing.Ready_LED == !$past(timing.Ready_LED)); // toggles
endproperty
```

D.2.2. Motion Detection Properties

The specification for the motion detection has two statements. We break out each part of those statements into six properties. This module can be verified using just the inputs and outputs listed in Table 11.

Table 11. Signals used in Verification of the Motion Detection Module

Signal	Type
Push_button_reset	Input
Material_alarm_A	Input
Material_alarm_B	Input
AtoB_alarm	Output
BtoA_alarm	Output

D.2.2.1. At Reset, Both Alarms are High

The direct way of writing this fails as the reset is synchronous.

```
property P1a_alarms_high_at_reset;
   disable iff (1'b0) // Don't disable this check at reset
   (motion_detection.Reset == 1'b0)
      |->
      motion_detection.AtoB_alarm &&
      motion_detection.BtoA_alarm;
endproperty
```

The property passes if the alarms are asserted are the next clock cycle after reset.

```
property P1b_alarms_high_at_reset_after_1clk;
   disable iff (1'b0) // Don't disable this check at reset
   (motion_detection.Reset == 1'b0)
      |=>
      motion_detection.AtoB_alarm &&
      motion_detection.BtoA_alarm;
endproperty
```

D.2.2.2. Alarms Disabled upon Push_button_reset

When Push_button_reset is asserted, both alarms are negated (on the next clock cycle).

```
property P2b_push_button_reset_low;
   !motion_detection.push_button_reset |=>
      !(motion_detection.AtoB_alarm || motion_detection.BtoA_alarm);
endproperty
```

D.2.2.3. Generation of AtoB_alarm

```
property P3_AtoB_alarm_SPEC;
   $rose(motion_detection.push_button_reset) ##1 // rising edge
   motion_detection.push_button_reset throughout
      ( (!motion_detection.Material_alarm_B &&
      !motion_detection.Material_alarm_A )[*] ##1
      motion_detection.Material_alarm_A // Until material Alarm goes high
      ##1 // sometime later B goes high
      motion_detection.Material_alarm_B[->1])
         |=>
         motion_detection.AtoB_alarm;
endproperty
```

D.2.2.4. No Additional Alarm if AtoB_alarm is Asserted

Ensure that once AtoB_alarm has gone asserted, BtoA_larm won't go assert before any push button reset. To avoid a problem at reset, we state that there has to be at least one clock cycle previously where Push_button_reset is negated.

```
property P4_AtoB_not_in_between_BtoA;
   $rose(motion_detection.push_button_reset) ##1
   ((motion_detection.push_button_reset && !motion_detection.BtoA_alarm) throughout
      (motion_detection.AtoB_alarm)[->1])
         |->
         (!motion_detection.BtoA_alarm throughout
            !motion_detection.push_button_reset[->1]);
endproperty
```

D.2.2.5. Generation of BtoA_alarm

```
property P5_BtoA_alarm;
   $rose(motion_detection.push_button_reset) ##1 // rising edge
   motion_detection.push_button_reset throughout
      ( (!motion_detection.Material_alarm_A &&
      !motion_detection.Material_alarm_B )[*] ##1
      motion_detection.Material_alarm_B && !motion_detection.Material_alarm_A // Until material Alarm
      ##1 // sometime later A goes high
      motion_detection.Material_alarm_A[->1])
         |=>
         motion_detection.BtoA_alarm;
 endproperty
```

D.2.2.6. No Additional Alarm if BtoA_alarm is Set

```
property P6_BtoA_not_in_between_AtoB;
   $rose(motion_detection.push_button_reset) ##1
   ((motion_detection.push_button_reset && !motion_detection.AtoB_alarm) throughout
      (motion_detection.BtoA_alarm)[->1])
         |->
         (!motion_detection.AtoB_alarm throughout
            !motion_detection.push_button_reset[->1]);
endproperty
```

D.2.3. Detector Properties

Table 12. Signals used in Verification of the Detector Module

Signal	Type
Push_button_reset	Input
Pulse	Input
Clear_pulse_counter	Input
Detector_ready	Output
Material_alarm	Output
High_background_alarm	Output
Low_background_alarm	Output
avg.Pause	External
alarms.Material_alarm	External

D.2.3.1. Pausing the Averager

After the rising edge of material_alarm, pause the averager.

```
property P1_det_pause_avg_at_alarm;
```

```
   $rose(detector.alarms.Material_alarm) |=> detector.avg.Pause;
endproperty
```

D.2.3.2. Unpausing the Averager

After the falling edge of the material_alarm, unpause the averager.

```
property P2_det_unpause_avg_after_alarm;
   $fell(detector.alarms.Material_alarm) |=> !detector.avg.Pause;
endproperty
```

D.2.3.3. Uninterrupted Operation of the Averager

Without an edge on material_alarm, make no change in pausing the averager.

```
property P3_det_hold_pause_stable_if_no_alarm_edge;
   $stable(detector.alarms.Material_alarm) |=> $stable(detector.avg.Pause);
endproperty
```

D.2.3.4. Signal Connectivity of the Detector

Ensure direct connections of wires.

```
property P4_detector_sigs_connected;
   // Averager signals
   (detector.avg.Load == detector.Clear_pulse_counter) &&
   (detector.avg.Pulse_count == detector.counter.Count) &&
   // counter signals
   (detector.counter.Pulse == detector.Pulse) &&
   (detector.counter.Clear == detector.Clear_pulse_counter) &&
   // alarms signals
   (detector.alarms.Push_button_reset == detector.Push_button_reset) &&
    (detector.alarms.Average_valid == detector.avg.Valid) &&
    (detector.alarms.Sigma_valid == detector.sqrt.Done) &&
    (detector.alarms.Sigma == detector.sqrt.root ) &&
    (detector.alarms.Count == detector.counter.Count) &&
    (detector.alarms.Average == detector.avg.Average) &&
    //sqrt: Done and root are output
    (detector.sqrt.Start == detector.Clear_pulse_counter) &&
    (detector.sqrt.a == detector.avg.Average) &&
    // Detector top
    (detector.Detector_ready == detector.avg.Valid) &&
    (detector.Material_alarm == detector.alarms.Material_alarm) &&
    (detector.High_background_alarm == detector.alarms.High_background_alarm) &&
    (detector.Low_background_alarm == detector.alarms.Low_background_alarm);
 endproperty // P3_detector_sigs_connected
```

D.2.4. Pulse Counter Properties

Table 13. Signals used in Verification of the Pulse Counter Module

Signal	Type
Clear	Input
Pulse	Input
Count	Output

D.2.4.1. Clearing the Count

Assertion of clear zeroes the count.

```
property P1_count_clear;
   pulse_counter.Clear |=> pulse_counter.Count == 0;
endproperty
```

D.2.4.2. Holding the Count

With no activity on pulse or clear, there is no change in count.

```
property P2_count_no_change_prop;
   !pulse_counter.Pulse && !pulse_counter.Clear
      |->
      ##1 $stable(pulse_counter.Count);
endproperty
```

D.2.4.3. Counter Saturation

At max count, the counter saturates and does not rollover.

```
property P3_count_saturation;
   !pulse_counter.Clear && // As long as Clear is low
   (pulse_counter.Count == 'h3ff) // and count is max
      |=> (pulse_counter.Count == 'h3ff); // count stays at max
endproperty
```

D.2.4.4. Increment the Count

If clear is negated, the logic detects a pulse, and the pulse counter is not saturated, then increment the count on the next clock cycle.

```
property P4_count_increments_prop;
   $rose(pulse_counter.Pulse) &&
   !pulse_counter.Clear &&
   (pulse_counter.Count != 'h3ff)
      |->
      ##1 pulse_counter.Count == ($past(pulse_counter.Count) + 1);
endproperty
```

D.2.5. Alarm Processor Properties

Table 14. Signals used in Verification of the Alarm Processor Module

Signal	**Type**
Push_button_reset	Input
Average	Input
Average_valid	Input
Count	Input
Sigma	Input
High_background_alarm	Output
Low_background_alarm	Output
Material_alarm	Output

D.2.5.1. Triggering the High Background Alarm

The High_background_alarm asserts when:

- Average >= HIGH_ALARM_LEVEL
- Average_valid asserted
- Push_button_reset negated

```
property P1_high_alarm;
  alarm_processor.Average_valid &&
  (alarm_processor.Average >= 30) &&
  alarm_processor.Push_button_reset
     |=>
     alarm_processor.High_background_alarm;
endproperty
```

D.2.5.2. Triggering the Low Background Alarm

The Low background_alarm asserts when:

- Average <= LOW_ALARM_LEVEL
- Average_valid asserted
- Push_button_reset negated

```
property P2_low_alarm;
  alarm_processor.Average_valid &&
  (alarm_processor.Average <= 5) &&
  alarm_processor.Push_button_reset
     |=>
     alarm_processor.Low_background_alarm;
endproperty
```

D.2.5.3. Triggering the Material Alarm

The Material_alarm asserts when:

- Count > (4*Sigma + Average)
- Average_valid asserted
- Sigma_valid asserted
- Push_button_reset negated

```
property P3_material_alarm;
  alarm_processor.Average_valid &&
  alarm_processor.sigma_valid &&
  (int'(alarm_processor.Count) >
  (($unsigned(alarm_processor.Sigma) <<< 2) +
  $unsigned(alarm_processor.Average)))
  &&
  alarm_processor.Push_button_reset
     |=>
     alarm_processor.Material_alarm;
endproperty
```

D.2.5.4. Alarms cannot trigger during Reset

High/Low/Material alarms can only assert if Push_button_reset is negated.

```
property P4_alarm_iff_pushbutton;
  alarm_processor.High_background_alarm ||
  alarm_processor.Low_background_alarm ||
  alarm_processor.Material_alarm |->
  $past(alarm_processor.Push_button_reset);
endproperty
```

D.2.6. Count Averaging Properties

Table 15. Signals used in Verification of the Count Averaging Module

Signal	Type
Load	Input
Pause	Input
Pulse_count	Input
Valid	Output
Average	Output
dout	Internal
sum_r	Internal

D.2.6.1. Reset value of the average

At reset, the average is 0.

```
property P1_zero_avg_at_reset;
   disable iff (1’b0)
   (count_averaging.Reset == 1’b0) |=>
   count_averaging.Average == 1’b0;
endproperty
```

D.2.6.2. Computation of the Average

The average computation is performed when (Pause = 0) and (Load = 1).

1. If the average is as yet invalid, then Average = Average + Pulse_count

```
property P2a_avg_before_valid;
   (!count_averaging.Pause) &&
   count_averaging.Load &&
   !count_averaging.Valid
      |=>
      count_averaging.sum_r ==
      ($past(count_averaging.sum_r) + $past(count_averaging.Pulse_count));
endproperty
```

2. If the average is now valid, then Average = Average + Pulse_count - dout

```
property P2b_avg_when_valid;
   (!count_averaging.Pause) &&
   count_averaging.Load &&
   count_averaging.Valid
      |=>
      count_averaging.sum_r ==
      (($past(count_averaging.sum_r) +
      $past(count_averaging.Pulse_count)) -
      $past(count_averaging.dout) );
endproperty
```

3. The output value average is the lower resolution version of sum_r

```
property P3_avg_calc;
   count_averaging.Average ==
   count_averaging.sum_r[$high(count_averaging.sum_r) -: count_averaging.count_width];
endproperty
```

D.2.7. Sqrt Properties

The initial approach we took to verify this module was to validate the accurate implementation of the algorithm described in the specification (D.1.7). Upon consultation with the designer, the approach taken instead was to model the input/output relationship as given in the equation.

Table 16. Signals used in Verification of the Sqrt Module

Signal	Type
a	Input
Start	Input
Root	Output
Done	Output

D.2.7.1. Reset Value of the Done Signal

At Reset, Done is asserted. Note, this assertion only works one clock cycle after reset (synchronous).

```
property P1_sqrt_at_reset;
   disable iff (1'b0)
   (sqrt.Reset == 1'b0)
      |=>
      sqrt.Done == 1'b1;
endproperty
```

D.2.7.2. Complete Square Root Computation

This property snapshots the value of the input “a” at the "Start" step and check to see if the output value has the correct square root.

```
property P2_sqrt_complete;
   //var type(sqrt.a) inputval;
   var logic[9:0] inputval;
   (sqrt.Done && $rose(sqrt.Start), inputval=sqrt.a) ##1 sqrt.Done[->1]
      |->
      (sqrt.root * sqrt.root) <= (inputval) &&
      (sqrt.root+1)*(sqrt.root+1) > inputval;
endproperty // P5_sqrt_complete
```

D.2.7.3. Example of a Square Root Computation

This property explicitly shows a non-trivial example of square root computation. This step is performed as a sanity check.

```
property P3_sqrt_cover;
   sqrt.Done && $rose(sqrt.Start) && (sqrt.a > 4) ##1 sqrt.Done[->1];
endproperty
```

D.2.8. SP_mem Properties

SP_mem is a single port memory that can be both written to and read from at any time. For the memory, we expect that once an address in memory is written with a data value, on the next read of that same address, the original data value will be retrieved.

Table 17. Signals used in Verification of the SP_mem Module

Signal	Type
Addr	Input
Wr	Input
Din	Input
Dout	Output

D.2.8.1. Memory Retrieval

The property is written by taking a snapshot of the address and value during a write and checking that the returned value at the next read is the value that was captured. These properties can only be verified for a small memory size (less than 64 elements).

```
property P1_sp_mem_retrieval;
   var logic[sp_mem.data_width-1:0] mem_write_val;
   var logic[sp_mem.addr_width-1:0] mem_write_addr;
   (sp_mem.Wr, mem_write_val = sp_mem.Din, mem_write_addr = sp_mem.Addr)
   ##1 (!(sp_mem.Addr == mem_write_addr && sp_mem.Wr)) // No more writes to the same address
   throughout
   ((sp_mem.Addr == mem_write_addr)[->1:$]) // Until some future read (could be many reads)
      |=> (mem_write_val == sp_mem.Dout);
endproperty
```

The following is a sanity check forced example showing memory write and read.

```
property P2_sp_mem_example;
   var logic[sp_mem.data_width-1:0] mem_write_val;
   var logic[sp_mem.addr_width-1:0] mem_write_addr;
   (sp_mem.Wr &&
   sp_mem.mem[0] == 1 &&
   sp_mem.mem[1] == 2 &&
   sp_mem.mem[2] ==3 &&
   sp_mem.mem[3] == 3 &&
   sp_mem.Din==7,
   mem_write_val = sp_mem.Din, mem_write_addr = sp_mem.Addr)
   ##1 (sp_mem.Addr != mem_write_addr)[*4] // different address for 4 cycles
   ##1 (sp_mem.Addr == mem_write_addr && !sp_mem.Wr) // read from that address
   ##1 (sp_mem.Addr != mem_write_addr && sp_mem.Wr) // write to different address
   ##1 (!(sp_mem.Addr == mem_write_addr && sp_mem.Wr))
   throughout
   ((sp_mem.Addr == mem_write_addr)[->1])
      |=> (mem_write_val == sp_mem.Dout);
 endproperty
```

D.2.8.2. Alternative Properties for Memory

We provide some alternate properties for memory that are easier to verify for larger memories. But the disadvantage is that they are less complete and depend upon the memory internals. These steps are broken up into the following three properties. Of these properties, the memory write and memory read properties will verify in a reasonable amount of time for larger-sized memories. Proving the memory stability property in a reasonable amount of time remains a challenge, but can be done for the sizes of memory used in this design.

1. Memory write: If the write signal asserts, then on the next clock cycle, the value of Din is stored at the address given in Addr.

```
property P3_sp_mem_write;
   sp_mem.Wr |=> sp_mem.mem[$past(sp_mem.Addr)] == $past(sp_mem.Din);
endproperty
```

2. Memory read: Dout on the next clock cycle provides the value of the memory at the address provided by Addr on the current cycle.

```
property P4_sp_mem_read;
   var logic[sp_mem.data_width-1:0] data_expec;
   (1'b1, data_expec = sp_mem.mem[sp_mem.Addr]) ##1 (sp_mem.Dout == data_expec);
endproperty
```

3. Memory stability: If there is no write to a specific memory address, then the memory value at that address is unaltered.

```
reg[sp_mem.addr_width-1:0] mem_addr_chk; // Free register
asm1: assume property($stable(mem_addr_chk)); // Force this to be stable for the check below
property P5_sp_mem_stable;
   var logic [sp_mem.data_width-1:0] mem_val;
   // Not writing to memory at address mem_addr_chk,
   // Capture the value at mem_addr_chk
   (!(sp_mem.Wr && mem_addr_chk == sp_mem.Addr), mem_val=sp_mem.mem[mem_addr_chk] )
      |=> sp_mem.mem[mem_addr_chk] == mem_val; // No change in value at mem_addr_chk
endproperty
```

D.2.9. Top Level Properties

Top level verification consists of verifying the module inter-connectivity and of verifying the proper aggregation of signals from the two detectors.

D.2.9.1. High Background Alarm

```
property P1_top_high_background_alarm;
   top_level.High_background_alarm ==
   (top_level.detector_A.High_background_alarm ||
   top_level.detector_B.High_background_alarm);
endproperty // P1_top_high_background_alarm
```

D.2.9.2. Low Background Alarm

```
property P2_top_low_background_alarm;
   top_level.Low_background_alarm ==
   (top_level.detector_A.Low_background_alarm ||
   top_level.detector_B.Low_background_alarm);
endproperty // P2_top_low_background_alarm
```

D.2.9.3. Timer is Enabled Only When Both Detectors Are Ready

```
property P3_top_timer_enabled_when_detectors_ready;
   top_level.timer.Detector_ready == (top_level.detector_A.Detector_ready &&
   top_level.detector_B.Detector_ready);
endproperty // P3_top_timer_enabled_when_detectors_ready
```

D.2.9.4. Connectivity at the Top Level

```
property P4_top_level_signal_connectivity;
   // sync
   top_level.Reset == top_level.sync.Reset &&
   top_level.Push_button_reset == top_level.sync.Push_button_reset &&
   top_level.Pulse_A == top_level.sync.Pulse_A &&
   top_level.Pulse_B == top_level.sync.Pulse_B &&
   // timer already taken care off.
   // detector_A
   top_level.detector_A.Push_button_reset == top_level.sync.Push_button_reset_sync &&
   top_level.detector_A.Pulse == top_level.sync.Pulse_A_sync &&
   top_level.detector_A.Clear_pulse_counter == top_level.timer.Clear_pulse_counter &&
   // detector_B
   top_level.detector_B.Push_button_reset == top_level.sync.Push_button_reset_sync &&
   top_level.detector_B.Pulse == top_level.sync.Pulse_B_sync &&
   top_level.detector_B.Clear_pulse_counter == top_level.timer.Clear_pulse_counter &&
   // motion
   top_level.motion.Push_button_reset == top_level.sync.Push_button_reset_sync &&
   top_level.motion.Material_alarm_A == top_level.detector_A.Material_alarm &&
   top_level.motion.Material_alarm_B == top_level.detector_B.Material_alarm &&
   // top level output
   top_level.AtoB_alarm == top_level.motion.AtoB_alarm &&
   top_level.BtoA_alarm == top_level.motion.BtoA_alarm;
 endproperty
```